CyPari2 Documentation

Release 2.1.3

Many People

Mar 23, 2023

CONTENTS:

1	Interface to the PARI library									
	1.1 Guide to real precision in the PARI interface	2								
2	The Gen class wrapping PARI's GEN type	383								
3	Memory management for Gens on the PARI stack or the heap	769								
4	Convert Python functions to PARI closures	771								
5	Handling PARI errors	773								
6	Convert PARI objects to/from Python/C native types	775								
7	Indices and tables	781								
Py	Python Module Index									
In	Index									

ONE

INTERFACE TO THE PARI LIBRARY

AUTHORS:

- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-03-06): added newtonpoly
- Justin Walker: contributed some of the function definitions
- Gonzalo Tornaria: improvements to conversions; much better error handling.
- Robert Bradshaw, Jeroen Demeyer, William Stein (2010-08-15): Upgrade to PARI 2.4.3 (Sage ticket #9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (Sage ticket #11611, Sage ticket #11854, Sage ticket #11952)
- Peter Bruin (2013-11-17): split off this file from gen.pyx (Sage ticket #15185)
- Jeroen Demeyer (2014-02-09): upgrade to PARI 2.7 (Sage ticket #15767)
- Jeroen Demeyer (2014-09-19): upgrade to PARI 2.8 (Sage ticket #16997)
- Jeroen Demeyer (2015-03-17): automatically generate methods from pari.desc (Sage ticket #17631 and Sage ticket #17860)
- Luca De Feo (2016-09-06): Separate Sage-specific components from generic C-interface in Pari (Sage ticket #20241)

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari('5! + 10/x')
(120*x + 10)/x
>>> pari('intnum(x=0,13,sin(x)+sin(x^2) + x)')
85.6215190762676
>>> f = pari('x^3 - 1')
>>> v = f.factor(); v
[x - 1, 1; x^2 + x + 1, 1]
>>> v[0] # indexing is 0-based unlike in GP.
[x - 1, x^2 + x + 1]~
>>> v[1]
[1, 1]~
```

For most functions, you can call the function as method of pari or you can first create a Gen object and then call the function as method of that. In other words, the following two commands do the same:

>>> pari('x^3 - 1').factor()
[x - 1, 1; x^2 + x + 1, 1]
>>> pari.factor('x^3 - 1')
[x - 1, 1; x^2 + x + 1, 1]

Arithmetic operations cause all arguments to be converted to PARI:

```
>>> type(pari(1) + 1)
<... 'cypari2.gen.Gen'>
>>> type(1 + pari(1))
<... 'cypari2.gen.Gen'>
```

1.1 Guide to real precision in the PARI interface

In the PARI interface, "real precision" refers to the precision of real numbers, so it is the floating-point precision. This is a non-trivial issue, since there are various interfaces for different things.

1.1.1 Internal representation of floating-point numbers in PARI

Real numbers in PARI have a precision associated to them, which is always a multiple of the CPU wordsize. So, it is a multiple of 32 of 64 bits. When converting a float from Python to PARI, the float has 53 bits of precision which is rounded up to 64 bits in PARI:

```
>>> x = 1.0
>>> pari(x)
1.000000000000
>>> pari(x).bitprecision()
64
```

It is possible to change the precision of a PARI object with the Gen.bitprecision() method:

```
>>> p = pari(1.0)
>>> p.bitprecision()
64
>>> p = p.bitprecision(100)
>>> p.bitprecision() # Rounded up to a multiple of the wordsize
128
```

Beware that these extra bits are just bogus. For example, this will not magically give a more precise approximation of math.pi:

```
>>> import math
>>> p = pari(math.pi)
>>> pari("Pi") - p
1.225148... E-16
>>> p = p.bitprecision(1000)
>>> pari("Pi") - p
1.225148... E-16
```

Another way to create numbers with many bits is to use a string with many digits:

```
>>> p = pari("3.1415926535897932384626433832795028842")
>>> p.bitprecision()
128
```

1.1.2 Output precision for printing

Even though PARI reals have a precision, not all significant bits are printed by default. The maximum number of digits when printing a PARI real can be set using the methods *Pari.set_real_precision_bits()* or *Pari.set_real_precision()*. Note that this will also change the input precision for strings, see *Input precision for func-tion calls*.

We create a very precise approximation of pi and see how it is printed in PARI:

```
>>> pi = pari.pi(precision=1024)
```

The default precision is 15 digits:

```
>>> pi
3.14159265358979
```

With a different precision, we see more digits. Note that this does not affect the object **pi** at all, it only affects how it is printed:

```
>>> _ = pari.set_real_precision(50)
>>> pi
3.1415926535897932384626433832795028841971693993751
```

Back to the default:

```
>>> _ = pari.set_real_precision(15)
>>> pi
3.14159265358979
```

1.1.3 Input precision for function calls

When we talk about precision for PARI functions, we need to distinguish three kinds of calls:

- 1. Using the string interface, for example pari("sin(1)").
- 2. Using the library interface with *exact* inputs, for example pari.sin(1).
- 3. Using the library interface with *inexact* inputs, for example pari.sin(1.0).

In the first case, the relevant precision is the one set by the methods *Pari.set_real_precision_bits()* or *Pari.set_real_precision()*:

```
>>> pari.set_real_precision_bits(150)
>>> pari("sin(1)")
0.841470984807896506652502321630298999622563061
>>> pari.set_real_precision_bits(53)
>>> pari("sin(1)")
0.841470984807897
```

In the second case, the precision can be given as the argument precision in the function call, with a default of 53 bits. The real precision set by *Pari.set_real_precision_bits()* or *Pari.set_real_precision()* does not affect the call (but it still affects printing).

As explained before, the precision increases to a multiple of the wordsize (and you should not assume that the extra bits are meaningful):

```
>>> a = pari.sin(1, precision=180); a
0.841470984807897
>>> a.bitprecision()
192
>>> b = pari.sin(1, precision=40); b
0.841470984807897
>>> b.bitprecision()
64
>>> c = pari.sin(1); c
0.841470984807897
>>> c.bitprecision()
64
>>> pari.set_real_precision_bits(90)
>>> print(a); print(b); print(c)
0.841470984807896506652502322
0.8414709848078965067
0.8414709848078965067
```

In the third case, the precision is determined only by the inexact inputs and the precision argument is ignored:

Tests:

Check that the documentation is generated correctly:

```
>>> from inspect import getdoc
>>> getdoc(pari.Pi)
'The constant :math:`\\pi` ...'
```

Check that output from PARI's print command is actually seen by Python (Sage ticket #9636):

```
>>> pari('print("test")')
test
```

Verify that nfroots() (which has an unusual signature with a non-default argument following a default argument) works:

```
>>> pari.nfroots(x='x^4 - 1')
[-1, 1]
>>> pari.nfroots(pari.nfinit('t^2 + 1'), "x^4 - 1")
[-1, 1, Mod(-t, t^2 + 1), Mod(t, t^2 + 1)]
```

Reset default precision for the following tests:

```
>>> pari.set_real_precision_bits(53)
```

Test that interrupts work properly:

```
>>> pari.allocatemem(8000000, 2**29)
PARI stack size set to 8000000 bytes, maximum size set to ...
>>> from cysignals.alarm import alarm, AlarmInterrupt
>>> for i in range(1, 11):
... try:
... alarm(i/11.0)
... pari.binomial(2**100, 2**22)
... except AlarmInterrupt:
... pass
```

Test that changing the stack size using default works properly:

```
>>> pari.default("parisizemax", 2**23)
>>> pari = cypari2.Pari() # clear stack
>>> a = pari(1)
>>> pari.default("parisizemax", 2**29)
>>> a + a
2
>>> pari.default("parisizemax")
536870912
```

class cypari2.pari_instance.Pari

List(x)

Create an empty list or convert *x* to a list.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.List(range(5))
List([0, 1, 2, 3, 4])
>>> L = pari.List()
>>> L
List([])
>>> L.listput(42, 1)
42
>>> L
List([42])
>>> L.listinsert(24, 1)
24
>>> L
List([24, 42])
```

allocatemem(s, sizemax, *, silent)

Change the PARI stack space to the given size s (or double the current size if s is 0) and change the maximum stack size to sizemax.

PARI tries to use only its current stack (the size which is set by s), but it will increase its stack if needed up to the maximum size which is set by sizemax.

The PARI stack is never automatically shrunk. You can use the command pari.allocatemem(10^6) to reset the size to 10^6, which is the default size at startup. Note that the results of computations using cypari are copied to the Python heap, so they take up no space in the PARI stack. The PARI stack is cleared after every computation.

It does no real harm to set this to a small value as the PARI stack will be automatically enlarged when we run out of memory.

INPUT:

- s an integer (default: 0). A non-zero argument is the size in bytes of the new PARI stack. If *s* is zero, double the current stack size.
- sizemax an integer (default: 0). A non-zero argument is the maximum size in bytes of the PARI stack. If sizemax is 0, the maximum of the current maximum and s is taken.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.allocatemem(10**7, 10**7)
PARI stack size set to 10000000 bytes, maximum size set to 100...
>>> pari.allocatemem() # Double the current size
PARI stack size set to 20000000 bytes, maximum size set to 200...
>>> pari.stacksize()
20000000
>>> pari.allocatemem(10**6)
PARI stack size set to 1000000 bytes, maximum size set to 200...
```

The following computation will automatically increase the PARI stack size:

>>> a = pari('2^100000000')

a is now a Python variable on the Python heap and does not take up any space on the PARI stack. The PARI stack is still large because of the computation of a:

```
>>> pari.stacksize() > 10**6
True
```

Setting a small maximum size makes this fail:

```
>>> pari.allocatemem(10**6, 2**22)
PARI stack size set to 1000000 bytes, maximum size set to 4194304
>>> a = pari('2^100000000')
Traceback (most recent call last):
....
PariError: _^s: the PARI stack overflows (current size: 10000000; maximum size:_
....
Parierror: _^s: the pari.allocatemem() to change the stack size and try again
```

Tests:

Do the same without using the string interface and starting from a very small stack size:

```
>>> pari.allocatemem(1, 2**26)
PARI stack size set to 1024 bytes, maximum size set to 67108864
>>> a = pari(2)**100000000
```

```
>>> pari.stacksize() > 10**6
True
```

We do not allow sizemax less than s:

```
>>> pari.allocatemem(10**7, 10**6)
Traceback (most recent call last):
...
ValueError: the maximum size (10000000) should be at least the stack size_
$\top$(1000000)
```

complex(re, im)

Create a new complex number, initialized from re and im.

debugstack()

Print the internal PARI variables top (top of stack), avma (available memory address, think of this as the stack pointer), bot (bottom of stack).

euler(precision)

Euler's constant $\gamma = 0.57721...$ Note that Euler is one of the few reserved names which cannot be used for user variables.

factorial_int(n)

Return the factorial of the integer n as a PARI gen. Give result as an integer.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.factorial_int(0)
1
>>> pari.factorial_int(1)
1
>>> pari.factorial_int(5)
120
>>> pari.factorial_int(25)
15511210043330985984000000
```

genus2red(P, p)

Let *P* be a polynomial with integer coefficients. Determines the reduction of the (proper, smooth) genus 2 curve *C/QQ*, defined by the hyperelliptic equation $y^2 = P$. The special syntax genus2red([P,Q]) is also allowed, where the polynomials *P* and *Q* have integer coefficients, to represent the model $y^2 + Q(x)y = P(x)$.

If the second argument p is specified, it must be a prime. Then only the local information at p is computed and returned.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> x = pari('x')
>>> pari.genus2red([-5*x**5, x**3 - 2*x**2 - 2*x + 1])
[1416875, [2, -1; 5, 4; 2267, 1], ..., [[2, [2, [Mod(1, 2)]], []], [5, [1, []],...]
```

```
→ ["[V] page 156", [3]]], [2267, [2, [Mod(432, 2267)]], ["[I{1-0-0}] page 170", ]
→ []]]]
>>> pari.genus2red([-5*x**5, x**3 - 2*x**2 - 2*x + 1],2267)
[2267, Mat([2267, 1]), ..., [2267, [2, [Mod(432, 2267)]], ["[I{1-0-0}] page 170
→ ", []]]]
```

get_debug_level()

Set the debug PARI C library variable.

get_real_precision()

Returns the current PARI default real precision.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new PARI objects in CyPari2. It has *no* effect on the precision of computations within the pari library.

See also:

get_real_precision_bits() to get the precision in bits.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.get_real_precision()
15
```

get_real_precision_bits()

Return the current PARI default real precision in bits.

This is used both for creation of new objects from strings and for printing. It determines the number of digits in which real numbers numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new PARI objects using cypari. It has *no* effect on the precision of computations within the PARI library.

See also:

get_real_precision() to get the precision in decimal digits.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.get_real_precision_bits()
53
```

init_primes(M)

Recompute the primes table including at least all primes up to M (but possibly more).

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.init_primes(200000)
```

We make sure that ticket Sage ticket #11741 has been fixed:

```
>>> pari.init_primes(2**30)
Traceback (most recent call last):
...
ValueError: Cannot compute primes beyond 436273290
```

matrix(m, n, entries)

matrix(long m, long n, entries=None): Create and return the m x n PARI matrix with given list of entries.

Examples:

>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.matrix(3, 3, range(9))
[0, 1, 2; 3, 4, 5; 6, 7, 8]

new_with_bits_prec(s, precision)

pari.new_with_bits_prec(self, s, precision) creates s as a PARI Gen with (at most) precision *bits* of precision.

one()

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.one()
1
```

static pari_version()

Return a string describing the version of PARI/GP.

```
>>> from cypari2 import Pari
>>> Pari.pari_version()
'GP/PARI CALCULATOR Version ...'
```

pi(precision)

The constant π (3.14159...). Note that Pi is one of the few reserved names which cannot be used for user variables.

polchebyshev(n, v)

Chebyshev polynomial of the first kind of degree *n*, in the variable *v*.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.polchebyshev(7)
64*x^7 - 112*x^5 + 56*x^3 - 7*x
>>> pari.polchebyshev(7, 'z')
64*z^7 - 112*z^5 + 56*z^3 - 7*z
>>> pari.polchebyshev(0)
1
```

polsubcyclo(*n*, *d*, *v*=*x*): return the pari list of polynomial(*s*)

defining the sub-abelian extensions of degree d of the cyclotomic field $QQ(zeta_n)$, where d divides phi(n).

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.polsubcyclo(8, 4)
[x^4 + 1]
>>> pari.polsubcyclo(8, 2, 'z')
[z^2 + 2, z^2 - 2, z^2 + 1]
>>> pari.polsubcyclo(8, 1)
[x - 1]
>>> pari.polsubcyclo(8, 3)
[]
```

primes(n, end)

Return a pari vector containing the first *n* primes, the primes in the interval [*n*, *end*], or the primes up to *end*.

INPUT:

Either

```
• n – integer
```

or

• n – list or tuple [a, b] defining an interval of primes

or

• n, end – start and end point of an interval of primes

or

- end end point for the list of primes
- OUTPUT: a PARI list of prime numbers

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.primes(3)
[2, 3, 5]
>>> pari.primes(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
>>> pari.primes(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
>>> len(pari.primes(1000))
1000
>>> pari.primes(11,29)
[11, 13, 17, 19, 23, 29]
>>> pari.primes((11,29))
[11, 13, 17, 19, 23, 29]
>>> pari.primes(end=29)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
>>> pari.primes(10**30, 10**30 + 100)
```

Tests:

```
>>> pari.primes(0)
[]
>>> pari.primes(-1)
[]
>>> pari.primes(end=1)
[]
>>> pari.primes(end=-1)
[]
>>> pari.primes(3,2)
[]
```

set_debug_level(level)

Set the debug PARI C library variable.

set_real_precision(n)

Sets the PARI default real precision in decimal digits.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new PARI objects in CyPari2. It has *no* effect on the precision of computations within the pari library.

Returns the previous PARI real precision.

See also:

set_real_precision_bits() to set the precision in bits.

Examples:

set_real_precision_bits(n)

Sets the PARI default real precision in bits.

This is used both for creation of new objects from strings and for printing. It determines the number of digits in which real numbers numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new PARI objects using cypari. It has *no* effect on the precision of computations within the PARI library.

See also:

set_real_precision() to set the precision in decimal digits.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.set_real_precision_bits(200)
>>> pari('1.2')
```

setrand(seed)

Sets PARI's current random number seed.

INPUT:

• seed - either a strictly positive integer or a GEN of type t_VECSMALL as output by getrand()

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.setrand(50)
>>> a = pari.getrand()
>>> pari.setrand(a)
>>> a == pari.getrand()
True
```

Tests:

Check that invalid inputs are handled properly:

```
>>> pari.setrand("foobar")
Traceback (most recent call last):
....
PariError: incorrect type in setrand (t_POL)
```

stacksize()

Return the current size of the PARI stack, which is 10⁶ by default. However, the stack size is automatically increased when needed up to the given maximum stack size.

See also:

- *stacksizemax()* to get the maximum stack size
- allocatemem() to change the current or maximum stack size

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.stacksize()
8000000
>>> pari.allocatemem(2**18, silent=True)
>>> pari.stacksize()
262144
```

stacksizemax()

Return the maximum size of the PARI stack, which is determined at startup in terms of available memory. Usually, the PARI stack size is (much) smaller than this maximum but the stack will be increased up to this maximum if needed.

See also:

- stacksize() to get the current stack size
- allocatemem() to change the current or maximum stack size

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.allocatemem(2**18, 2**26, silent=True)
>>> pari.stacksizemax()
67108864
```

vector(n, entries)

vector(long n, entries=None): Create and return the length n PARI vector with given list of entries.

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.vector(5, [1, 2, 5, 4, 3])
[1, 2, 5, 4, 3]
>>> pari.vector(2, ['x', 1])
[x, 1]
>>> pari.vector(2, ['x', 1, 5])
Traceback (most recent call last):
...
IndexError: length of entries (=3) must equal n (=2)
```

version()

Return the PARI version as tuple with 3 or 4 components: (major, minor, patch) or (major, minor, patch, VCSversion).

Examples:

```
>>> from cypari2 import Pari
>>> Pari().version() >= (2, 9, 0)
True
```

zero()

Examples:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> pari.zero()
0
```

class cypari2.pari_instance.Pari_auto

Part of the Pari class containing auto-generated functions.

You must never use this class directly (in fact, Python may crash if you do), use the derived class Pari instead.

Catalan(precision)

Catalan's constant $G = \sum_{n>=0} ((-1)^n)/((2n+1)^2) = 0.91596...$ Note that Catalan is one of the few reserved names which cannot be used for user variables.

Col(x, n)

Transforms the object x into a column vector. The dimension of the resulting vector can be optionally specified via the extra parameter n.

If n is omitted or 0, the dimension depends on the type of x; the vector has a single component, except when x is

- a vector or a quadratic form (in which case the resulting vector is simply the initial object considered as a row vector),
- a polynomial or a power series. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree. In this last case, Vec is the reciprocal function of Pol and Ser respectively,
- a matrix (the column of row vector comprising the matrix is returned),
- a character string (a vector of individual characters is returned).

In the last two cases (matrix and character string), n is meaningless and must be omitted or an error is raised. Otherwise, if n is given, 0 entries are appended at the end of the vector if n > 0, and prepended at the beginning if n < 0. The dimension of the resulting vector is ||n||.

See ??Vec for examples.

Colrev(x, n)

As Col(x, -n), then reverse the result. In particular, Colrev is the reciprocal function of Polrev: the coefficients of the vector start with the constant coefficient of the polynomial and the others follow by increasing degree.

Euler(precision)

Euler's constant $\gamma = 0.57721...$ Note that Euler is one of the few reserved names which cannot be used for user variables.

I()

The complex number $\sqrt{-1}$.

List(x)

Transforms a (row or column) vector x into a list, whose components are the entries of x. Similarly for a list, but rather useless in this case. For other types, creates a list with the single element x.

Map(x)

A "Map" is an associative array, or dictionary: a data type composed of a collection of (key, value) pairs, such that each key appears just once in the collection. This function converts the matrix $[a_1, b_1; a_2, b_2; ...; a_n, b_n]$ to the map $a_i : -- > b_i$.

? M = Map(factor(13!)); ? mapget(M,3) %2 = 5

If the argument x is omitted, creates an empty map, which may be filled later via mapput.

Mat(x)

Transforms the object x into a matrix. If x is already a matrix, a copy of x is created. If x is a row (resp. column) vector, this creates a 1-row (resp. 1-column) matrix, *unless* all elements are column (resp. row) vectors of the same length, in which case the vectors are concatenated sideways and the attached big matrix is returned. If x is a binary quadratic form, creates the attached $2x^2$ matrix. Otherwise, this creates a $1x^1$ matrix containing x.

```
? Mat(x + 1)
%1 =
[x + 1]
? Vec( matid(3) )
\%2 = [[1, 0, 0] \sim, [0, 1, 0] \sim, [0, 0, 1] \sim]
? Mat(%)
%3 =
[1 \ 0 \ 0]
[0 1 0]
[0 0 1]
? Col( [1,2; 3,4] )
%4 = [[1, 2], [3, 4]]~
? Mat(%)
%5 =
[1 2]
[3 4]
? Mat(Qfb(1,2,3))
%6 =
[1 1]
[1 3]
```

Mod(a, b)

In its basic form, create an intmod or a polmod (*amodb*); *b* must be an integer or a polynomial. We then obtain a t_INTMOD and a t_POLMOD respectively:

? t = Mod(2,17); t^8 %1 = Mod(1, 17) ? t = Mod(x,x^2+1); t^2 %2 = Mod(-1, x^2+1)

If a%b makes sense and yields a result of the appropriate type (t_INT or scalar/t_POL), the operation succeeds as well:

```
? Mod(1/2, 5)
%3 = Mod(3, 5)
? Mod(7 + 0(3^6), 3)
%4 = Mod(1, 3)
? Mod(Mod(1,12), 9)
%5 = Mod(1, 3)
? Mod(1/x, x^2+1)
%6 = Mod(-x, x^2+1)
? Mod(exp(x), x^4)
%7 = Mod(1/6*x^3 + 1/2*x^2 + x + 1, x^4)
```

If a is a complex object, "base change" it to $\mathbb{Z}/b\mathbb{Z}$ or K[x]/(b), which is equivalent to, but faster than, multiplying it by Mod(1,b):

```
? Mod([1,2;3,4], 2)
%8 =
```

```
[Mod(1, 2) Mod(0, 2)]
[Mod(1, 2) Mod(0, 2)]
? Mod(3*x+5, 2)
%9 = Mod(1, 2)*x + Mod(1, 2)
? Mod(x^2 + y*x + y^3, y^2+1)
%10 = Mod(1, y^2 + 1)*x^2 + Mod(y, y^2 + 1)*x + Mod(-y, y^2 + 1)
```

This function is not the same as $x \ % y$, the result of which has no knowledge of the intended modulus y. Compare

? x = 4 % 5; x + 1 %11 = 5 ? x = Mod(4,5); x + 1 %12 = Mod(0,5)

Note that such "modular" objects can be lifted via lift or centerlift. The modulus of a t_INTMOD or t_POLMOD z can be recovered via :math:`z.mod`.

Pi(precision)

The constant π (3.14159...). Note that Pi is one of the few reserved names which cannot be used for user variables.

Pol(*t*, *v*)

Transforms the object t into a polynomial with main variable v. If t is a scalar, this gives a constant polynomial. If t is a power series with nonnegative valuation or a rational function, the effect is similar to truncate, i.e. we chop off the $O(X^k)$ or compute the Euclidean quotient of the numerator by the denominator, then change the main variable of the result to v.

The main use of this function is when t is a vector: it creates the polynomial whose coefficients are given by t, with t[1] being the leading coefficient (which can be zero). It is much faster to evaluate Pol on a vector of coefficients in this way, than the corresponding formal expression $a_n X^n + ... + a_0$, which is evaluated naively exactly as written (linear versus quadratic time in n). Polrev can be used if one wants x[1] to be the constant coefficient:

```
? Pol([1,2,3])
%1 = x^2 + 2*x + 3
? Polrev([1,2,3])
%2 = 3*x^2 + 2*x + 1
```

The reciprocal function of Pol (resp. Polrev) is Vec (resp. Vecrev).

```
? Vec(Pol([1,2,3]))
%1 = [1, 2, 3]
? Vecrev( Polrev([1,2,3]) )
%2 = [1, 2, 3]
```

Warning. This is *not* a substitution function. It will not transform an object containing variables of higher priority than *v*.

```
? Pol(x + y, y)
 *** at top-level: Pol(x+y,y)
 *** ^------
 *** Pol: variable must have higher priority in gtopoly.
```

Polrev(t, v)

Transform the object t into a polynomial with main variable v. If t is a scalar, this gives a constant polynomial. If t is a power series, the effect is identical to truncate, i.e. it chops off the $O(X^k)$.

The main use of this function is when t is a vector: it creates the polynomial whose coefficients are given by t, with t[1] being the constant term. Pol can be used if one wants t[1] to be the leading coefficient:

```
? Polrev([1,2,3])
%1 = 3*x^2 + 2*x + 1
? Pol([1,2,3])
%2 = x^2 + 2*x + 3
```

The reciprocal function of Pol (resp. Polrev) is Vec (resp. Vecrev).

Qfb(*a*, *b*, *c*, *D*, *precision*)

Creates the binary quadratic form $ax^2 + bxy + cy^2$. If $b^2 - 4ac > 0$, initialize Shanks' distance function to D. Negative definite forms are not implemented, use their positive definite counterpart instead.

Ser(*s*, *v*, *d*, *serprec*)

Transforms the object s into a power series with main variable v (x by default) and precision (number of significant terms) equal to $d \ge 0$ (d = series precision by default). If s is a scalar, this gives a constant power series in v with precision d. If s is a polynomial, the polynomial is truncated to d terms if needed

```
? \ps
seriesprecision = 16 significant terms
? Ser(1) \\ 16 terms by default
%1 = 1 + 0(x^16)
? Ser(1, 'y, 5)
%2 = 1 + 0(y^5)
? Ser(x^2,, 5)
%3 = x^2 + 0(x^7)
? T = polcyclo(100)
%4 = x^40 - x^30 + x^20 - x^10 + 1
? Ser(T, 'x, 11)
%5 = 1 - x^10 + 0(x^11)
```

The function is more or less equivalent with multiplication by $1 + O(v^d)$ in these cases, only faster.

For the remaining types, vectors and power series, we first explain what occurs if d is omitted. In this case, the function uses exactly the amount of information given in the input:

- If s is already a power series in v, we return it verbatim;
- If s is a vector, the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (as in Polrev(x)); in other words we convert t_VEC / t_COL to the power series whose significant terms are exactly given by the vector entries.

On the other hand, if d is explicitly given, we abide by its value and return a series, truncated or extended with zeros as needed, with d significant terms.

```
? v = [1,2,3];
? Ser(v, t) \\ 3 terms: seriesprecision is ignored!
%7 = 1 + 2*t + 3*t^2 + 0(t^3)
? Ser(v, t, 7) \\ 7 terms as explicitly requested
%8 = 1 + 2*t + 3*t^2 + 0(t^7)
? s = 1+x+0(x^2);
? Ser(s)
```

```
%10 = 1 + x + 0(x^2) \ 2 \text{ terms: seriesprecision is ignored}
? Ser(s, x, 7) \\ extend to 7 terms
%11 = 1 + x + 0(x^7)
? Ser(s, x, 1) \\ truncate to 1 term
%12 = 1 + 0(x)
```

The warning given for Pol also applies here: this is not a substitution function.

Set(x)

Converts x into a set, i.e. into a row vector, with strictly increasing entries with respect to the (some-what arbitrary) universal comparison function cmp. Standard container types t_VEC, t_COL, t_LIST and t_VECSMALL are converted to the set with corresponding elements. All others are converted to a set with one element.

```
? Set([1,2,4,2,1,3])
%1 = [1, 2, 3, 4]
? Set(x)
%2 = [x]
? Set(Vecsmall([1,3,2,1,3]))
%3 = [1, 2, 3]
```

Strchr(x)

Deprecated alias for strchr.

Vec(*x*, *n*)

Transforms the object x into a row vector. The dimension of the resulting vector can be optionally specified via the extra parameter n. If n is omitted or 0, the dimension depends on the type of x; the vector has a single component, except when x is

- a vector or a quadratic form: returns the initial object considered as a row vector,
- a polynomial or a power series: returns a vector consisting of the coefficients. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree. In particular the valuation is ignored (which makes the function useful for series of negative valuation):

```
? Vec(3*x<sup>2</sup> + x)
%1 = [3, 1, 0]
? Vec(x<sup>2</sup> + 3*x<sup>3</sup> + 0(x<sup>5</sup>))
%2 = [1, 3, 0]
? Vec(x<sup>2</sup> + 3*x<sup>1</sup> + 0(x))
%3 = [1, 3, 0]
```

Vec is the reciprocal function of Pol for a polynomial and of Ser for power series of valuation 0.

• a matrix: returns the vector of columns comprising the matrix,

? m = [1,2,3;4,5,6] %4 = [1 2 3] [4 5 6] ? Vec(m) %5 = [[1, 4]~, [2, 5]~, [3, 6]~] • a character string: returns the vector of individual characters,

```
? Vec("PARI")
%6 = ["P", "A", "R", "I"]
```

- a map: returns the vector of the domain of the map,
- an error context (t_ERROR): returns the error components, see iferr.

In the last four cases (matrix, character string, map, error), n is meaningless and must be omitted or an error is raised. Otherwise, if n is given, 0 entries are appended at the end of the vector if n > 0, and prepended at the beginning if n < 0. The dimension of the resulting vector is ||n||. This allows to write a conversion function for series that takes positive valuations into account:

```
? serVec(s) = Vec(s, -serprec(s,variable(s)));
? Vec(x^2 + 3*x^3 + 0(x^5))
%2 = [0, 0, 1, 3, 0]
```

(That function is not intended for series of negative valuation.)

Vecrev(x, n)

As Vec(x, -n), then reverse the result. In particular, Vecrev is the reciprocal function of Polrev: the coefficients of the vector start with the constant coefficient of the polynomial and the others follow by increasing degree.

Vecsmall(x, n)

Transforms the object x into a row vector of type t_VECSMALL. The dimension of the resulting vector can be optionally specified via the extra parameter n.

This acts as Vec(x, n), but only on a limited set of objects: the result must be representable as a vector of small integers. If x is a character string, a vector of individual characters in ASCII encoding is returned (strchr yields back the character string).

abs(*x*, *precision*)

Absolute value of x (modulus if x is complex). Rational functions are not allowed. Contrary to most transcendental functions, an exact argument is *not* converted to a real number before applying abs and an exact result is returned if possible.

```
? abs(-1)
%1 = 1
? abs(3/7 + 4/7*I)
%2 = 5/7
? abs(1 + I)
%3 = 1.414213562373095048801688724
```

If x is a polynomial, returns -x if the leading coefficient is real and negative else returns x. For a power series, the constant coefficient is considered instead.

acos(*x*, *precision*)

Principal branch of $\cos^{-1}(x) = -i \log(x + i\sqrt{1-x^2})$. In particular, $\Re(acos(x)) \in [0, \pi]$ and if $x \in \mathbb{R}$ and ||x|| > 1, then acos(x) is complex. The branch cut is in two pieces:] - oo, -1], continuous with quadrant II, and [1, +oo[, continuous with quadrant IV. We have $acos(x) = \pi/2 - asin(x)$ for all x.

acosh(*x*, *precision*)

Principal branch of $\cosh^{-1}(x) = 2\log(\sqrt{(x+1)/2} + \sqrt{(x-1)/2})$. In particular, $\Re(acosh(x)) \ge 0$ and $\Im(acosh(x)) \in]-\pi,\pi]$; if $x \in \mathbb{R}$ and x < 1, then acosh(x) is complex.

addhelp(sym, str)

Changes the help message for the symbol sym. The string *str* is expanded on the spot and stored as the online help for sym. It is recommended to document global variables and user functions in this way, although gp will not protest if you don't.

You can attach a help text to an alias, but it will never be shown: aliases are expanded by the ? help operator and we get the help of the symbol the alias points to. Nothing prevents you from modifying the help of built-in PARI functions. But if you do, we would like to hear why you needed it!

Without addhelp, the standard help for user functions consists of its name and definition.

gp> f(x) = x^2; gp> ?f f = (x)->x^2

Once addhelp is applied to f, the function code is no longer included. It can still be consulted by typing the function name:

```
gp> addhelp(f, "Square")
gp> ?f
Square
gp> f
%2 = (x)->x^2
```

addprimes(x)

Adds the integers contained in the vector x (or the single integer x) to a special table of "user-defined primes", and returns that table. Whenever factor is subsequently called, it will trial divide by the elements in this table. If x is empty or omitted, just returns the current list of extra primes.

The entries in x must be primes: there is no internal check, even if the factor_proven default is set. To remove primes from the list use removeprimes.

```
agm(x, y, precision)
```

Arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the optimal AGM is returned (the largest in absolute value over all choices of the signs of the square roots). p-adic or power series arguments are also allowed. Note that a p-adic agm exists only if x/y is congruent to 1 modulo p (modulo 16 for p = 2). x and y cannot both be vectors or matrices.

```
airy(z, precision)
```

Airy [Ai, Bi] functions of argument z.

```
? [A,B] = airy(1);
? A
```

```
%2 = 0.13529241631288141552414742351546630617
? B
%3 = 1.2074235949528712594363788170282869954
```

algadd(al, x, y)

Given two elements x and y in al, computes their sum x + y in the algebra al.

? A = alginit(nfinit(y),[-1,1]); ? algadd(A,[1,0]~,[1,2]~) %2 = [2, 2]~

Also accepts matrices with coefficients in al.

algalgtobasis(al, x)

Given an element x in the central simple algebra al output by alginit, transforms it to a column vector on the integral basis of al. This is the inverse function of algbasistoalg.

```
? A = alginit(nfinit(y<sup>2-5</sup>),[2,y]);
? algalgtobasis(A,[y,1]~)
%2 = [0, 2, 0, -1, 2, 0, 0, 0]~
? algbasistoalg(A,algalgtobasis(A,[y,1]~))
%3 = [Mod(Mod(y, y<sup>2</sup> - 5), x<sup>2</sup> - 2), 1]~
```

algaut(al)

Given a cyclic algebra $al = (L/K, \sigma, b)$ output by alginit, returns the automorphism σ .

? nf = nfinit(y); ? p = idealprimedec(nf,7)[1]; ? p2 = idealprimedec(nf,11)[1]; ? A = alginit(nf,[3,[[p,p2],[1/3,2/3]],[0]]); ? algaut(A) %5 = -1/3*x^2 + 1/3*x + 26/3

algb(al)

Given a cyclic algebra $al = (L/K, \sigma, b)$ output by alginit, returns the element $b \in K$.

```
nf = nfinit(y);
? p = idealprimedec(nf,7)[1];
? p2 = idealprimedec(nf,11)[1];
? A = alginit(nf,[3,[[p,p2],[1/3,2/3]],[0]]);
? algb(A)
%5 = Mod(-77, y)
```

algbasis(al)

Given a central simple algebra *al* output by alginit, returns a \mathbb{Z} -basis of the order O_0 stored in *al* with respect to the natural order in *al*. It is a maximal order if one has been computed.

A = alginit(nfinit(y), [-1,-1]); ? algbasis(A) %2 = [1 0 0 1/2]

[0]	1	0	1/2]								
[0]	0	1	1/2]								
[0]	0	0	1/2]								

algbasistoalg(al, x)

Given an element *x* in the central simple algebra *al* output by alginit, transforms it to its algebraic representation in *al*. This is the inverse function of algalgtobasis.

? A = alginit(nfinit(y^2-5),[2,y]); ? z = algbasistoalg(A,[0,1,0,0,2,-3,0,0]~); ? liftall(z) %3 = [(-1/2*y - 2)*x + (-1/4*y + 5/4), -3/4*y + 7/4]~ ? algalgtobasis(A,z) %4 = [0, 1, 0, 0, 2, -3, 0, 0]~

algcenter(al)

If *al* is a table algebra output by algtableinit, returns a basis of the center of the algebra *al* over its prime field (\mathbb{Q} or \mathbb{F}_p). If *al* is a central simple algebra output by alginit, returns the center of *al*, which is stored in *al*.

A simple example: the $2x^2$ upper triangular matrices over \mathbb{Q} , generated by I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$: the diagonal matrices form the center.

? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]]; ? A = algtableinit(mt); ? algcenter(A) \\ = (I_2) %3 = [1] [0] [0]

An example in the central simple case:

```
? nf = nfinit(y^3-y+1);
? A = alginit(nf, [-1,-1]);
? algcenter(A).pol
%3 = y^3 - y + 1
```

algcentralproj(al, z, maps)

Given a table algebra al output by algtableinit and a t_VEC $z = [z_1, ..., z_n]$ of orthogonal central idempotents, returns a t_VEC $[al_1, ..., al_n]$ of algebras such that $al_i = z_i al$. If maps = 1, each al_i is a t_VEC [quo, proj, lift] where quo is the quotient algebra, proj is a t_MAT representing the projection onto this quotient and lift is a t_MAT representing a lift.

A simple example: $\mathbb{F}_2 x \mathbb{F}_4$, generated by 1 = (1, 1), e = (1, 0) and x such that $x^2 + x + 1 = 0$. We have $e^2 = e$, $x^2 = x + 1$ and ex = 0.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
```

```
? e = [0,1,0]~;
? e2 = algsub(A,[1,0,0]~,e);
? [a,a2] = algcentralproj(A,[e,e2]);
? algdim(a)
%6 = 1
? algdim(a2)
%7 = 2
```

algchar(al)

Given an algebra *al* output by alginit or algtableinit, returns the characteristic of *al*.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,13);
? algchar(A)
%3 = 13
```

algcharpoly(al, b, v, abs)

Given an element b in al, returns its characteristic polynomial as a polynomial in the variable v. If al is a table algebra output by algtableinit or if abs = 1, returns the absolute characteristic polynomial of b, which is an element of $\mathbb{F}_p[v]$ or $\mathbb{Q}[v]$; if al is a central simple algebra output by alginit and abs = 0, returns the reduced characteristic polynomial of b, which is an element of K[v] where K is the center of al.

```
? al = alginit(nfinit(y), [-1,-1]); \\ (-1,-1)_Q
? algcharpoly(al, [0,1]~)
%2 = x^2 + 1
? algcharpoly(al, [0,1]~,,1)
%3 = x^4 + 2*x^2 + 1
? nf = nfinit(y^2-5);
? al = alginit(nf,[-1,y]);
? a = [y,1+x]~*Mod(1,y^2-5)*Mod(1,x^2+1);
? P = lift(algcharpoly(al,a))
%7 = x^2 - 2*y*x + (-2*y + 5)
? algcharpoly(al,a,,1)
%8 = x^8 - 20*x^6 - 80*x^5 + 110*x^4 + 800*x^3 + 1500*x^2 - 400*x + 25
? lift(P*subst(P,y,-y)*Mod(1,y^2-5))^2
%9 = x^8 - 20*x^6 - 80*x^5 + 110*x^4 + 800*x^3 + 1500*x^2 - 400*x + 25
```

Also accepts a square matrix with coefficients in al.

algdegree(al)

Given a central simple algebra *al* output by alginit, returns the degree of *al*.

? nf = nfinit(y^3-y+1); ? A = alginit(nf, [-1,-1]); ? algdegree(A) %3 = 2

algdep(z, k, flag)

z being real/complex, or p-adic, finds a polynomial (in the variable 'x) of degree at most k, with integer coefficients, having z as approximate root. Note that the polynomial which is obtained is not necessarily the "correct" one. In fact it is not even guaranteed to be irreducible. One can check the closeness either by

a polynomial evaluation (use subst), or by computing the roots of the polynomial given by algdep (use polroots or polrootspadic).

Internally, $lindep([1, z, ..., z^k], flag)$ is used. A nonzero value of flag may improve on the default behavior if the input number is known to a *huge* accuracy, and you suspect the last bits are incorrect: if flag > 0 the computation is done with an accuracy of flag decimal digits; to get meaningful results, the parameter flag should be smaller than the number of correct decimal digits in the input. But default values are usually sufficient, so try without flag first:

? \p200 ? z = 2^(1/6)+3^(1/5); ? algdep(z, 30); \\ right in 280ms ? algdep(z, 30, 100); \\ wrong in 169ms ? algdep(z, 30, 170); \\ right in 288ms ? algdep(z, 30, 200); \\ wrong in 320ms ? \p250 ? z = 2^(1/6)+3^(1/5); \\ recompute to new, higher, accuracy ! ? algdep(z, 30); \\ right in 329ms ? algdep(z, 30, 200); \\ right in 324ms ? \p500 ? algdep(2^(1/6)+3^(1/5), 30); \\ right in 677ms ? \p1000 ? algdep(2^(1/6)+3^(1/5), 30); \\ right in 1.5s

The changes in realprecision only affect the quality of the initial approximation to $2^{1/6} + 3^{1/5}$, algdep itself uses exact operations. The size of its operands depend on the accuracy of the input of course: more accurate input means slower operations.

Proceeding by increments of 5 digits of accuracy, algdep with default flag produces its first correct result at 195 digits, and from then on a steady stream of correct results:

```
\\ assume T contains the correct result, for comparison
forstep(d=100, 250, 5, localprec(d);\
print(d, " ", algdep(2^{(1/6)}+3^{(1/5)},30) == T))
```

The above example is the test case studied in a 2000 paper by Borwein and Lisonek: Applications of integer relation algorithms, *Discrete Math.*, **217**, p. 65–82. The version of PARI tested there was 1.39, which succeeded reliably from precision 265 on, in about 200 as much time as the current version.

algdim(al, abs)

If *al* is a table algebra output by algtableinit or if abs = 1, returns the dimension of *al* over its prime subfield (\mathbb{Q} or \mathbb{F}_p). If *al* is a central simple algebra output by alginit and abs = 0, returns the dimension of *al* over its center.

? nf = nfinit(y³-y+1); ? A = alginit(nf, [-1,-1]); ? algdim(A) %3 = 4 ? algdim(A,1) %4 = 12

algdisc(al)

Given a central simple algebra *al* output by alginit, computes the discriminant of the order O_0 stored in *al*, that is the determinant of the trace form $\text{Tr} : O_0 x O_0 \to \mathbb{Z}$.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-3,1-y]);
? [PR,h] = alghassef(A)
%3 = [[[2, [2, 0]~, 1, 2, 1], [3, [3, 0]~, 1, 2, 1]], Vecsmall([0, 1])]
? n = algdegree(A);
? D = algdim(A,1);
? h = vector(#h, i, n - gcd(n,h[i]));
? n^D * nf.disc^(n^2) * idealnorm(nf, idealfactorback(nf,PR,h))^n
%4 = 12960000
? algdisc(A)
%5 = 12960000
```

algdivl(*al*, *x*, *y*)

Given two elements x and y in al, computes their left quotient $x \setminus y$ in the algebra al: an element z such that xz = y (such an element is not unique when x is a zerodivisor). If x is invertible, this is the same as $x^{-1}y$. Assumes that y is left divisible by x (i.e. that z exists). Also accepts matrices with coefficients in al.

algdivr(al, x, y)

Given two elements x and y in al, returns xy^{-1} . Also accepts matrices with coefficients in al.

alggroup(gal, p)

Initializes the group algebra K[G] over $K = \mathbb{Q}$ (p omitted) or \mathbb{F}_p where G is the underlying group of the galoisinit structure gal. The input gal is also allowed to be a t_VEC of permutations that is closed under products.

Example:

? K = nfsplitting(x^3-x+1); ? gal = galoisinit(K); ? al = alggroup(gal); ? algissemisimple(al) %4 = 1 ? G = [Vecsmall([1,2,3]), Vecsmall([1,3,2])]; ? al2 = alggroup(G, 2); ? algissemisimple(al2) %8 = 0

alggroupcenter(gal, p, cc)

Initializes the center Z(K[G]) of the group algebra K[G] over $K = \mathbb{Q}$ (p = 0 or omitted) or \mathbb{F}_p where G is the underlying group of the galoisinit structure gal. The input gal is also allowed to be a t_VEC of permutations that is closed under products. Sets cc to a t_VEC [elts, conjclass, rep, flag] where elts is a sorted t_VEC containing the list of elements of G, conjclass is a t_VECSMALL of the same length as elts containing the index of the conjugacy class of the corresponding element (an integer between 1 and the number of conjugacy classes), and rep is a t_VECSMALL of length the number of conjugacy classes giving for each conjugacy class the index in elts of a representative of this conjugacy class. Finally flag is 1 if and only if the permutation representation of G is transitive, in which case the *i*-th element of elts is characterized by g[1] = i; this is always the case when gal is a galoisinit structure. The basis of Z(K[G]) as output consists of the indicator functions of the conjugacy classes in the ordering given by cc. Example:

? K = nfsplitting(x^4+x+1); ? gal = galoisinit(K); \\ S4 ? al = alggroupcenter(gal,,&cc); ? algiscommutative(al)

```
%4 = 1
? #cc[3] \\ number of conjugacy classes of S4
%5 = 5
? gal = [Vecsmall([1,2,3]),Vecsmall([1,3,2])]; \\ C2
? al = alggroupcenter(gal,,&cc);
? cc[3]
%8 = Vecsmall([1, 2])
? cc[4]
%9 = 0
```

alghasse(al, pl)

Given a central simple algebra *al* output by alginit and a prime ideal or an integer between 1 and $r_1 + r_2$, returns a t_FRAC *h* : the local Hasse invariant of *al* at the place specified by *pl*.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? alghasse(A, 1)
%3 = 1/2
? alghasse(A, 2)
%4 = 0
? alghasse(A, idealprimedec(nf,2)[1])
%5 = 1/2
? alghasse(A, idealprimedec(nf,5)[1])
%6 = 0
```

alghassef(al)

Given a central simple algebra *al* output by alginit, returns a t_VEC $[PR, h_f]$ describing the local Hasse invariants at the finite places of the center: PR is a t_VEC of primes and h_f is a t_VECSMALL of integers modulo the degree *d* of *al*. The Hasse invariant of *al* at the primes outside PR is 0, but PR can include primes at which the Hasse invariant is 0.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,2*y-1]);
? [PR,hf] = alghassef(A);
? PR
%4 = [[19, [10, 2]~, 1, 1, [-8, 2; 2, -10]], [2, [2, 0]~, 1, 2, 1]]
? hf
%5 = Vecsmall([1, 0])
```

alghassei(al)

Given a central simple algebra al output by alginit, returns a t_VECSMALL h_i of r_1 integers modulo the degree d of al, where r_1 is the number of real places of the center: the local Hasse invariants of al at infinite places.

? nf = nfinit(y^2-5); ? A = alginit(nf, [-1,y]); ? alghassei(A) %3 = Vecsmall([1, 0])

algindex(al, pl)

Returns the index of the central simple algebra A over K (as output by alginit), that is the degree e of the unique central division algebra D over K such that A is isomorphic to some matrix algebra $M_k(D)$. If pl

is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case return the local index at the place *pl* instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algindex(A, 1)
%3 = 2
? algindex(A, 2)
%4 = 1
? algindex(A, idealprimedec(nf,2)[1])
%5 = 2
? algindex(A, idealprimedec(nf,5)[1])
%6 = 1
? algindex(A)
%7 = 2
```

alginit(B, C, v, maxord)

Initializes the central simple algebra defined by data B, C and variable v, as follows.

• (multiplication table) B is the base number field K in nfinit form, C is a "multiplication table" over K. As a K-vector space, the algebra is generated by a basis $(e_1 = 1, ..., e_n)$; the table is given as a t_VEC of n matrices in $M_n(K)$, giving the left multiplication by the basis elements e_i , in the given basis. Assumes that $e_1 = 1$, that the multiplication table is integral, and that $(\bigoplus_{i=1}^n Ke_i, C)$ describes a central simple algebra over K.

```
{ mi = [0,-1,0, 0;
1, 0,0, 0;
0, 0,0,-1;
0, 0,1, 0];
mj = [0, 0,-1,0;
0, 0, 0,1;
1, 0, 0,0;
0,-1, 0,0];
mk = [0, 0, 0, 0;
0, 0,-1, 0;
0, 1, 0, 0;
1, 0, 0,-1];
A = alginit(nfinit(y), [matid(4), mi,mj,mk], 0); }
```

represents (in a complicated way) the quaternion algebra $(-1, -1)_{\mathbb{Q}}$. See below for a simpler solution.

• (cyclic algebra) *B* is an rnf structure attached to a cyclic number field extension L/K of degree *d*, *C* is a t_VEC [sigma,b] with 2 components: sigma is a t_POLMOD representing an automorphism generating Gal(L/K), *b* is an element in K^* . This represents the cyclic algebra $(L/K, \sigma, b)$. Currently the element *b* has to be integral.

```
? Q = nfinit(y); T = polcyclo(5, 'x); F = rnfinit(Q, T);
? A = alginit(F, [Mod(x^2,T), 3]);
```

defines the cyclic algebra $(L/\mathbb{Q}, \sigma, 3)$, where $L = \mathbb{Q}(\zeta_5)$ and $\sigma : \zeta : --- > \zeta^2$ generates $Gal(L/\mathbb{Q})$.

• (quaternion algebra, special case of the above) B is an nf structure attached to a number field K, C = [a, b] is a vector containing two elements of K^* with a not a square in K, returns the quaternion algebra $(a, b)_K$. The variable v ('x by default) must have higher priority than the variable of K.pol and is used to represent elements in the splitting field $L = K[x]/(x^2 - a)$.

? Q = nfinit(y); A = alginit(Q, [-1,-1]); \\ (-1,-1)_Q

• (algebra/K defined by local Hasse invariants) B is an nf structure attached to a number field $K, C = [d, [PR, h_f], h_i]$ is a triple containing an integer d > 1, a pair $[PR, h_f]$ describing the Hasse invariants at finite places, and h_i the Hasse invariants at archimedean (real) places. A local Hasse invariant belongs to $(1/d)\mathbb{Z}/\mathbb{Z} \subset \mathbb{Q}/\mathbb{Z}$, and is given either as a t_FRAC (lift to $(1/d)\mathbb{Z})$, a t_INT or t_INTMOD modulo d (lift to $\mathbb{Z}/d\mathbb{Z}$); a whole vector of local invariants can also be given as a t_VECSMALL, whose entries are handled as t_INT s. PR is a list of prime ideals (prid structures), and h_f is a vector of the same length giving the local invariants at those maximal ideals. The invariants at infinite real places are indexed by the real roots K.roots: if the Archimedean place v is attached to the j-th root, the value of h_v is given by $h_i[j]$, must be 0 or 1/2 (or d/2 modulo d), and can be nonzero only if d is even.

By class field theory, provided the local invariants h_v sum to 0, up to Brauer equivalence, there is a unique central simple algebra over K with given local invariants and trivial invariant elsewhere. In particular, up to isomorphism, there is a unique such algebra A of degree d.

We realize A as a cyclic algebra through class field theory. The variable v ('x by default) must have higher priority than the variable of K.pol and is used to represent elements in the (cyclic) splitting field extension L/K for A.

```
? nf = nfinit(y<sup>2</sup>+1);
? PR = idealprimedec(nf,5); #PR
%2 = 2
? hi = [];
? hf = [PR, [1/3,-1/3]];
? A = alginit(nf, [3,hf,hi]);
? algsplittingfield(A).pol
%6 = x<sup>3</sup> - 21*x + 7
```

• (matrix algebra, toy example) B is an nf structure attached to a number field K, C = d is a positive integer. Returns a cyclic algebra isomorphic to the matrix algebra $M_d(K)$.

In all cases, this function computes a maximal order for the algebra by default, which may require a lot of time. Setting maxord = 0 prevents this computation.

The pari object representing such an algebra A is a t_VEC with the following data:

- A splitting field L of A of the same degree over K as A, in rnfinit format, accessed with algsplittingfield.
- The Hasse invariants at the real places of K, accessed with alghassei.
- The Hasse invariants of A at the finite primes of K that ramify in the natural order of A, accessed with alghassef.
- A basis of an order O_0 expressed on the basis of the natural order, accessed with algbasis.
- A basis of the natural order expressed on the basis of O_0 , accessed with alginvbasis.
- The left multiplication table of O_0 on the previous basis, accessed with algmultable.
- The characteristic of A (always 0), accessed with algchar.
- The absolute traces of the elements of the basis of O_0 .
- If A was constructed as a cyclic algebra $(L/K, \sigma, b)$ of degree d, a t_VEC $[\sigma, \sigma^2, ..., \sigma^{d-1}]$. The function algaut returns σ .
- If A was constructed as a cyclic algebra $(L/K, \sigma, b)$, the element b, accessed with algb.

- If A was constructed with its multiplication table mt over K, the t_VEC of t_MAT mt, accessed with algrelmultable.
- If A was constructed with its multiplication table mt over K, a t_VEC with three components: a t_COL representing an element of A generating the splitting field L as a maximal subfield of A, a t_MAT representing an L-basis B of A expressed on the Z-basis of O₀, and a t_MAT representing the Z-basis of O₀ expressed on B. This data is accessed with algsplittingdata.

alginv(al, x)

Given an element x in al, computes its inverse x^{-1} in the algebra al. Assumes that x is invertible.

? A = alginit(nfinit(y), [-1,-1]); ? alginv(A,[1,1,0,0]~) %2 = [1/2, 1/2, 0, 0]~

Also accepts matrices with coefficients in al.

alginvbasis(al)

Given an central simple algebra *al* output by alginit, returns a \mathbb{Z} -basis of the natural order in *al* with respect to the order O_0 stored in *al*.

```
A = alginit(nfinit(y), [-1,-1]);
? alginvbasis(A)
%2 =
[1 0 0 -1]
[0 1 0 -1]
[0 0 1 -1]
[0 0 0 2]
```

algisassociative(mt, p)

Returns 1 if the multiplication table mt is suitable for algtableinit(mt,p), 0 otherwise. More precisely, mt should be a t_VEC of n matrices in $M_n(K)$, giving the left multiplications by the basis elements $e_1, ..., e_n$ (structure constants). We check whether the first basis element e_1 is 1 and $e_i(e_je_k) = (e_ie_j)e_k$ for all i, j, k.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? algisassociative(mt)
%2 = 1
```

May be used to check a posteriori an algebra: we also allow mt as output by algtableinit (p is ignored in this case).

algiscommutative(al)

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is commutative.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algiscommutative(A)
%3 = 0
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
```

```
? algiscommutative(A)
%6 = 1
```

algisdivision(al, pl)

Given a central simple algebra al output by alginit, tests whether al is a division algebra. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally a division algebra at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algisdivision(A, 1)
%3 = 1
? algisdivision(A, 2)
%4 = 0
? algisdivision(A, idealprimedec(nf,2)[1])
%5 = 1
? algisdivision(A, idealprimedec(nf,5)[1])
%6 = 0
? algisdivision(A)
%7 = 1
```

algisdivl(al, x, y, z)

Given two elements x and y in al, tests whether y is left divisible by x, that is whether there exists z in al such that xz = y, and sets z to this element if it exists.

```
? A = alginit(nfinit(y), [-1,1]);
? algisdivl(A,[x+2,-x-2]~,[x,1]~)
%2 = 0
? algisdivl(A,[x+2,-x-2]~,[-x,x]~,&z)
%3 = 1
? z
%4 = [Mod(-2/5*x - 1/5, x^2 + 1), 0]~
```

Also accepts matrices with coefficients in al.

algisinv(al, x, ix)

Given an element x in al, tests whether x is invertible, and sets ix to the inverse of x.

```
? A = alginit(nfinit(y), [-1,1]);
? algisinv(A,[-1,1]~)
%2 = 0
? algisinv(A,[1,2]~,&ix)
%3 = 1
? ix
%4 = [Mod(Mod(-1/3, y), x^2 + 1), Mod(Mod(2/3, y), x^2 + 1)]~
```

Also accepts matrices with coefficients in al.

algisramified(al, pl)

Given a central simple algebra al output by alginit, tests whether al is ramified, i.e. not isomorphic to a matrix algebra over its center. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally ramified at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algisramified(A, 1)
%3 = 1
? algisramified(A, 2)
%4 = 0
? algisramified(A, idealprimedec(nf,2)[1])
%5 = 1
? algisramified(A, idealprimedec(nf,5)[1])
%6 = 0
? algisramified(A)
%7 = 1
```

algissemisimple(al)

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is semisimple.

? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]]; ? A = algtableinit(mt); ? algissemisimple(A) %3 = 0 ? m_i=[0,-1,0,0;1,0,0,0;0,0,0,-1;0,0,1,0]; \\ quaternion algebra (-1,-1) ? m_j=[0,0,-1,0;0,0,0,1;1,0,0,0;0,-1,0,0]; ? m_k=[0,0,0,-1;0,0,-1,0;0,1,0,0;1,0,0,0]; ? mt = [matid(4), m_i, m_j, m_k]; ? A = algtableinit(mt); ? algissemisimple(A) %9 = 1

algissimple(al, ss)

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is simple. If ss = 1, assumes that the algebra *al* is semisimple without testing it.

```
? mt = [matid(3), [0,0,0;1,0,1;0,0,0], [0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt); \\ matrices [*,*; 0,*]
? algissimple(A)
%3 = 0
? algissimple(A,1) \\ incorrectly assume that A is semisimple
\%4 = 1
? m_i=[0,-1,0,0;1,0,0,0;0,0,0,-1;0,0,1,0];
? m_j=[0,0,-1,0;0,0,0,1;1,0,0,0;0,-1,0,0];
? m_k=[0,0,0,-1;0,0,b,0;0,1,0,0;1,0,0,0];
? mt = [matid(4), m_i, m_j, m_k];
? A = algtableinit(mt); \\ quaternion algebra (-1,-1)
? algissimple(A)
%10 = 1
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2); \\ direct product F_4 x F_2
? algissimple(A)
%13 = 0
```

algissplit(al, pl)

Given a central simple algebra al output by alginit, tests whether al is split, i.e. isomorphic to a matrix

algebra over its center. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally split at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algissplit(A, 1)
%3 = 0
? algissplit(A, 2)
%4 = 1
? algissplit(A, idealprimedec(nf,2)[1])
%5 = 0
? algissplit(A, idealprimedec(nf,5)[1])
%6 = 1
? algissplit(A)
%7 = 0
```

alglatadd(al, lat1, lat2, ptinter)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the sum *lat1* + *lat2*. If *ptinter* is present, set it to the intersection *lat1* \cap *lat2*.

? al = alginit(nfinit(y^2+7), [-1,-1]); ? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~); ? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~); ? latsum = alglatadd(al,lat1,lat2,&latinter); ? matdet(latsum[1]) %5 = 4 ? matdet(latinter[1]) %6 = 64

alglatcontains(al, lat, x, ptc)

Given an algebra al, a lattice *lat* and x in al, tests whether $x \in lat$. If *ptc* is present, sets it to the t_COL of coordinates of x in the basis of *lat*.

```
? al = alginit(nfinit(y<sup>2</sup>+7), [-1,-1]);
? al = [1,-1,0,1,2,0,1,2]~;
? lat1 = alglathnf(al,a1);
? alglatcontains(al,lat1,a1,&c)
%4 = 1
? c
%5 = [-1, -2, -1, 1, 2, 0, 1, 1]~
```

alglatelement(al, lat, c)

Given an algebra *al*, a lattice *lat* and a $t_COL c$, returns the element of *al* whose coordinates on the mathbb{Z}-basis of *lat* are given by *c*.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? a1 = [1,-1,0,1,2,0,1,2]~;
? lat1 = alglathnf(al,a1);
? c = [1..8]~;
? elt = alglatelement(al,lat1,c);
? alglatcontains(al,lat1,elt,&c2)
%6 = 1
? c==c2
%7 = 1
```
alglathnf(al, m, d)

Given an algebra al and a matrix m with columns representing elements of al, returns the lattice L generated by the columns of m. If provided, d must be a rational number such that L contains d times the natural basis of al. The argument m is also allowed to be a t_VEC of t_MAT, in which case m is replaced by the concatenation of the matrices, or a t_COL, in which case m is replaced by its left multiplication table as an element of al.

```
? al = alginit(nfinit(y<sup>2</sup>+7), [-1,-1]);
? a = [1,1,-1/2,1,1/3,-1,1,1]~;
? mt = algtomatrix(al,a,1);
? lat = alglathnf(al,mt);
? lat[2]
%5 = 1/6
```

alglatindex(al, lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the generalized index of *lat1* relative to *lat2*, i.e. $||lat2/lat1 \cap lat2||/||lat1/lat1 \cap lat2||$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? alglatindex(al,lat1,lat2)
%4 = 1
? lat1==lat2
%5 = 0
```

alglatinter(al, lat1, lat2, ptsum)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the intersection $lat1 \cap lat2$. If *ptsum* is present, sets it to the sum lat1 + lat2.

```
? al = alginit(nfinit(y<sup>2</sup>+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? latinter = alglatinter(al,lat1,lat2,&latsum);
? matdet(latsum[1])
%5 = 4
? matdet(latinter[1])
%6 = 64
```

alglatlefttransporter(al, lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the left transporter from *lat1* to *lat2*, i.e. the set of $x \in al$ such that $x.lat1 \subset lat2$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,-1,0,1,2,0,5,2]~);
? lat2 = alglathnf(al,[0,1,-2,-1,0,0,3,1]~);
? tr = alglatlefttransporter(al,lat1,lat2);
? a = alglatelement(al,tr,[0,0,0,0,0,0,0,1,0]~);
? alglatsubset(al,alglatmul(al,a,lat1),lat2)
%6 = 1
? alglatsubset(al,alglatmul(al,lat1,a),lat2)
%7 = 0
```

alglatmul(al, lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the lattice generated by the products of elements of *lat1* and *lat2*. One of *lat1* and *lat2* is also allowed to be an element of *al*; in this case, computes the product of the element and the lattice.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? a1 = [1,-1,0,1,2,0,1,2]~;
? a2 = [0,1,2,-1,0,0,3,1]~;
? lat1 = alglathnf(al,a1);
? lat2 = alglathnf(al,a2);
? lat3 = alglatmul(al,lat1,lat2);
? matdet(lat3[1])
%7 = 29584
? lat3 == alglathnf(al, algmul(al,a1,a2))
%8 = 0
? lat3 == alglatmul(al, lat1, a2)
%9 = 0
? lat3 == alglatmul(al, a1, lat2)
%10 = 0
```

alglatrighttransporter(al, lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the right transporter from *lat1* to *lat2*, i.e. the set of $x \in al$ such that $lat1.x \subset lat2$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,matdiagonal([1,3,7,1,2,8,5,2]));
? lat2 = alglathnf(al,matdiagonal([5,3,8,1,9,8,7,1]));
? tr = alglatrighttransporter(al,lat1,lat2);
? a = alglatelement(al,tr,[0,0,0,0,0,0,0,0,1]~);
? alglatsubset(al,alglatmul(al,lat1,a),lat2)
%6 = 1
? alglatsubset(al,alglatmul(al,a,lat1),lat2)
%7 = 0
```

alglatsubset(al, lat1, lat2, ptindex)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, tests whether $lat1 \subset lat2$. If it is true and *ptindex* is present, sets it to the index of *lat1* in *lat2*.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? alglatsubset(al,lat1,lat2)
%4 = 0
? latsum = alglatadd(al,lat1,lat2);
? alglatsubset(al,lat1,latsum,&index)
%6 = 1
? index
%7 = 4
```

algmakeintegral(mt, maps)

mt being a multiplication table over \mathbb{Q} in the same format as the input of algtableinit, computes an integral multiplication table *mt2* for an isomorphic algebra. When maps = 1, returns a t_VEC [*mt2*, *S*, *T*] where *S* and *T* are matrices respectively representing the map from the algebra defined by *mt* to the one defined by *mt2* and its inverse.

```
? mt = [matid(2), [0, -1/4; 1, 0]];
? algtableinit(mt);
*** at top-level: algtableinit(mt)
 *** ^_____
*** algtableinit: domain error in algtableinit: denominator(mt) != 1
? mt2 = algmakeintegral(mt);
? al = algtableinit(mt2);
? algisassociative(al)
%4 = 1
? [mt2, S, T] = algmakeintegral(mt,1);
? S
%6 =
[1 0]
[0 1/4]
? T
%7 =
[1 \ 0]
[0 4]
? vector(#mt, i, S * (mt * T[,i]) * T) == mt2
%8 = 1
```

algmul(al, x, y)

Given two elements x and y in al, computes their product xy in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algmul(A,[1,1,0,0]~,[0,0,2,1]~) %2 = [2, 3, 5, -4]~

Also accepts matrices with coefficients in al.

algmultable(al)

Returns a multiplication table of *al* over its prime subfield (\mathbb{Q} or \mathbb{F}_p), as a t_VEC of t_MAT: the left multiplication tables of basis elements. If *al* was output by algtableinit, returns the multiplication table used to define *al*. If *al* was output by alginit, returns the multiplication table of the order O_0 stored in *al*.

```
? A = alginit(nfinit(y), [-1,-1]);
? M = algmultable(A);
? #M
%3 = 4
? M[1] \\ multiplication by e_1 = 1
%4 =
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
? M[2]
%5 =
```

 $\begin{bmatrix} 0 & -1 & 1 & 0 \end{bmatrix}$ $\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}$ $\begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$ $\begin{bmatrix} 0 & 0 & -2 & -1 \end{bmatrix}$

algneg(al, x)

Given an element x in al, computes its opposite -x in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algneg(A,[1,1,0,0]~) %2 = [-1, -1, 0, 0]~

Also accepts matrices with coefficients in al.

algnorm(al, x, abs)

Given an element *x* in *al*, computes its norm. If *al* is a table algebra output by algtableinit or if abs = 1, returns the absolute norm of *x*, which is an element of \mathbb{F}_p of \mathbb{Q} ; if *al* is a central simple algebra output by alginit and abs = 0 (default), returns the reduced norm of *x*, which is an element of the center of *al*.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,19);
? algnorm(A, [0,-2,3]~)
%3 = 18
? nf = nfinit(y^2-5);
? B = alginit(nf,[-1,y]);
? b = [x,1]~;
? n = algnorm(B,b)
%7 = Mod(-y + 1, y^2 - 5)
? algnorm(B,b,1)
%8 = 16
? nfeltnorm(nf,n)^algdegree(B)
%9 = 16
```

Also accepts a square matrix with coefficients in al.

algpoleval(al, T, b)

Given an element b in al and a polynomial T in K[X], computes T(b) in al. Also accepts as input a t_VEC [b, mb] where mb is the left multiplication table of b.

```
? nf = nfinit(y^2-5);
? al = alginit(nf,[y,-1]);
? b = [1..8]~;
? pol = algcharpoly(al,b,,1);
? algpoleval(al,pol,b)==0
%5 = 1
? mb = algtomatrix(al,b,1);
? algpoleval(al,pol,[b,mb])==0
%7 = 1
```

algpow(*al*, *x*, *n*)

Given an element x in al and an integer n, computes the power x^n in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algpow(A,[1,1,0,0]~,7) %2 = [8, -8, 0, 0]~

Also accepts a square matrix with coefficients in al.

algprimesubalg(al)

al being the output of algtableinit representing a semisimple algebra of positive characteristic, returns a basis of the prime subalgebra of *al*. The prime subalgebra of *al* is the subalgebra fixed by the Frobenius automorphism of the center of *al*. It is abstractly isomorphic to a product of copies of \mathbb{F}_p .

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? algprimesubalg(A)
%3 =
[1 0]
[0 1]
[0 0]
```

algquotient(al, I, maps)

al being a table algebra output by algtableinit and *I* being a basis of a two-sided ideal of *al* represented by a matrix, returns the quotient al/I. When maps = 1, returns a t_VEC [al/I, proj, lift] where *proj* and *lift* are matrices respectively representing the projection map and a section of it.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? AQ = algquotient(A,[0;1;0]);
? algdim(AQ)
%4 = 2
```

algradical(al)

al being a table algebra output by algtableinit, returns a basis of the Jacobson radical of the algebra *al* over its prime field (\mathbb{Q} or \mathbb{F}_p).

Here is an example with $A = \mathbb{Q}[x]/(x^2)$, with the basis (1, x):

```
? mt = [matid(2),[0,0;1,0]];
? A = algtableinit(mt);
? algradical(A) \\ = (x)
%3 =
[0]
[1]
```

Another one with 2x2 upper triangular matrices over \mathbb{Q} , with basis I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$:

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algradical(A) \\ = (a)
%6 =
[0]
```

[1]		
[0]		

algramifiedplaces(al)

Given a central simple algebra al output by alginit, returns a t_VEC containing the list of places of the center of al that are ramified in al. Each place is described as an integer between 1 and r_1 or as a prime ideal.

? nf = nfinit(y^2-5); ? A = alginit(nf, [-1,y]); ? algramifiedplaces(A) %3 = [1, [2, [2, 0]~, 1, 2, 1]]

algrandom(al, b)

Given an algebra *al* and an integer *b*, returns a random element in *al* with coefficients in [-b, b].

algrelmultable(al)

Given a central simple algebra *al* output by alginit defined by a multiplication table over its center (a number field), returns this multiplication table.

```
? nf = nfinit(y^{3}-5); a = y; b = y^{2};
? \{m_i = [0,a,0,0];
 1,0,0,0;
 0,0,0,a;
 0, 0, 1, 0]; \}
? \{m_j = [0, 0, b, 0];
 0, 0,0,-b;
 1, 0,0, 0;
 0, -1, 0, 0]; \}
? \{m_k = [0, 0, 0, -a^*b;
 0, 0,b, 0;
 0,-a,0, 0;
1, 0, 0, 0];
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf,mt,'x);
? M = algrelmultable(A);
? M[2] == m_i
%8 = 1
? M[3] == m_j
\%9 = 1
? M[4] == m_k
\%10 = 1
```

algsimpledec(al, maps)

al being the output of algtableinit, returns a t_VEC $[J, [al_1, al_2, ..., al_n]]$ where J is a basis of the Jacobson radical of al and al/J is isomorphic to the direct product of the simple algebras al_i . When maps = 1, each al_i is replaced with a t_VEC $[al_i, proj_i, lift_i]$ where $proj_i$ and $lift_i$ are matrices respectively representing the projection map $al \rightarrow al_i$ and a section of it. Modulo J, the images of the $lift_i$ form a direct sum in al/J, so that the images of 1_i under $lift_i$ are central primitive idempotents of al/J. The factors are sorted by increasing dimension, then increasing dimension of the center. This ensures that the ordering of the isomorphism classes of the factors is deterministic over finite fields, but not necessarily over \mathbb{Q} .

algsplit(al, v)

If al is a table algebra over \mathbb{F}_p output by algtableinit that represents a simple algebra, computes an isomorphism between al and a matrix algebra $M_d(\mathbb{F}_{p^n})$ where $N = nd^2$ is the dimension of al. Returns a t_VEC [map, mapi], where:

- map is a t_VEC of N matrices of size dxd with t_FFELT coefficients using the variable v, representing the image of the basis of al under the isomorphism.
- mapi is an NxN matrix with t_INT coefficients, representing the image in al by the inverse isomorphism of the basis (b_i) of $M_d(\mathbb{F}_p[\alpha])$ (where α has degree n over \mathbb{F}_p) defined as follows: let $E_{i,j}$ be the matrix having all coefficients 0 except the (i, j)-th coefficient equal to 1, and define

```
b_{i_3+n(i_2+di_1)+1} = E_{i_1+1,i_2+1}\alpha^{i_3},
where : math : '0 <= i_1, i_2 < d'and : math : '0 <= i_3 < n'.
```

Example:

```
? al0 = alginit(nfinit(y^2+7), [-1,-1]);
? al = algtableinit(algmultable(al0), 3); \\ isomorphic to M_2(F_9)
? [map,mapi] = algsplit(al, 'a);
? x = [1,2,1,0,0,0,0,0]~; fx = map*x
%4 =
[2*a 0]
[ 0 2]
? y = [0,0,0,0,1,0,0,1]~; fy = map*y
%5 =
[1 2*a]
[2 a + 2]
? map*algmul(al,x,y) == fx*fy
%6 = 1
? map*mapi[,6]
%7 =
[0 0]
[a 0]
```

Warning. If *al* is not simple, algsplit(al) can trigger an error, but can also run into an infinite loop. Example:

```
? al = alginit(nfinit(y),[-1,-1]); \\ ramified at 2
? al2 = algtableinit(algmultable(al),2); \\ maximal order modulo 2
? algsplit(al2); \\ not semisimple, infinite loop
```

algsplittingdata(al)

Given a central simple algebra al output by alginit defined by a multiplication table over its center K (a number field), returns data stored to compute a splitting of al over an extension. This data is a t_VEC [t,Lbas,Lbasinv] with 3 components:

- an element t of al such that L = K(t) is a maximal subfield of al;
- a matrix Lbas expressing a *L*-basis of *al* (given an *L*-vector space structure by multiplication on the right) on the integral basis of *al*;
- a matrix Lbasinv expressing the integral basis of *al* on the previous *L*-basis.

```
? nf = nfinit(y^{3}-5); a = y; b = y^{2};
? \{m_i = [0,a,0,0;
1,0,0,0;
 0,0,0,a;
0, 0, 1, 0];
? \{m_j = [0, 0, b, 0;
 0, 0,0,-b;
 1, 0,0, 0;
 0, -1, 0, 0]; \}
? \{m_k = [0, 0, 0, -a^*b;
 0, 0,b, 0;
0,-a,0, 0;
1, 0, 0, 0];
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf,mt,'x);
? [t,Lb,Lbi] = algsplittingdata(A);
? t
\%8 = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]~;
? matsize(Lb)
%9 = [12, 2]
? matsize(Lbi)
%10 = [2, 12]
```

algsplittingfield(al)

Given a central simple algebra *al* output by alginit, returns an rnf structure: the splitting field of *al* that is stored in *al*, as a relative extension of the center.

```
nf = nfinit(y^{3}-5);
a = y; b = y^{2};
{m_i = [0, a, 0, 0;
1,0,0,0;
 0,0,0,a;
 0, 0, 1, 0]; \}
{m_j = [0, 0, b, 0;}
0, 0,0,-b;
 1, 0,0, 0;
 0, -1, 0, 0];
\{m_k = [0, 0, 0, -a^*b;
 0, 0,b, 0;
0,-a,0, 0;
 1, 0, 0, 0];
mt = [matid(4), m_i, m_j, m_k];
A = alginit(nf,mt, 'x);
algsplittingfield(A).pol
\%8 = x^{2} - y
```

algsqr(al, x)

Given an element x in al, computes its square x^2 in the algebra al.

```
? A = alginit(nfinit(y), [-1,-1]);
? algsqr(A,[1,0,2,0]~)
%2 = [-3, 0, 4, 0]~
```

Also accepts a square matrix with coefficients in al.

algsub(*al*, *x*, *y*)

Given two elements x and y in al, computes their difference x - y in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algsub(A,[1,1,0,0]~,[1,0,1,0]~) %2 = [0, 1, -1, 0]~

Also accepts matrices with coefficients in al.

algsubalg(al, B)

al being a table algebra output by algtableinit and *B* being a basis of a subalgebra of *al* represented by a matrix, computes an algebra *al2* isomorphic to *B*.

Returns [al2, B2] where B2 is a possibly different basis of the subalgebra al2, with respect to which the multiplication table of al2 is defined.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? B = algsubalg(A, [1,0; 0,0; 0,1]);
? algdim(A)
\%4 = 3
? algdim(B[1])
\%5 = 2
? m = matcompanion(x^{4+1});
? mt = [m^i | i <- [0..3]];
? al = algtableinit(mt);
? B = [1,0;0,0;0,1/2;0,0];
? al2 = algsubalg(al,B);
? algdim(al2[1])
? al2[2]
%13 =
[1 0]
[0 0]
[0 1]
[0 0]
```

algtableinit(mt, p)

Initializes the associative algebra over $K = \mathbb{Q}$ (p omitted) or \mathbb{F}_p defined by the multiplication table *mt*. As a K-vector space, the algebra is generated by a basis $(e_1 = 1, e_2, ..., e_n)$; the table is given as a t_VEC of n matrices in $M_n(K)$, giving the left multiplication by the basis elements e_i , in the given basis. Assumes that $e_1 = 1$, that $Ke_1 \oplus ... \oplus Ke_n$] describes an associative algebra over K, and in the case $K = \mathbb{Q}$ that the multiplication table is integral. If the algebra is already known to be central and simple, then the case $K = \mathbb{F}_p$ is useless, and one should use alginit directly.

The point of this function is to input a finite dimensional K-algebra, so as to later compute its radical, then to split the quotient algebra as a product of simple algebras over K.

The pari object representing such an algebra A is a t_VEC with the following data:

- The characteristic of A, accessed with algchar.
- The multiplication table of A, accessed with algmultable.
- The traces of the elements of the basis.

A simple example: the $2x^2$ upper triangular matrices over \mathbb{Q} , generated by I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$:

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algradical(A) \\ = (a)
%6 =
[0]
[1]
[0]
? algcenter(A) \\ = (I_2)
%7 =
[1]
[0]
[0]
```

algtensor(al1, al2, maxord)

Given two algebras *al1* and *al2*, computes their tensor product. Computes a maximal order by default. Prevent this computation by setting maxord = 0.

Currently only implemented for cyclic algebras of coprime degree over the same center K, and the tensor product is over K.

algtomatrix(al, x, abs)

Given an element x in al, returns the image of x under a homomorphism to a matrix algebra. If al is a table algebra output by algtableinit or if abs = 1, returns the left multiplication table on the integral basis; if al is a central simple algebra and abs = 0, returns $\phi(x)$ where $\phi : A \otimes_K L \to M_d(L)$ (where d is the degree of the algebra and L is an extension of L with [L : K] = d) is an isomorphism stored in al. Also accepts a square matrix with coefficients in al.

```
? A = alginit(nfinit(y), [-1,-1]);
? algtomatrix(A,[0,0,0,2]~)
%2 =
[Mod(x + 1, x^2 + 1) Mod(Mod(1, y)*x + Mod(-1, y), x^2 + 1)]
[Mod(x + 1, x^2 + 1) Mod(-x + 1, x^2 + 1)]
? algtomatrix(A,[0,1,0,0]~,1)
%2 =
[0 -1 1 0]
[1 0 1 1]
[0 0 -2 -1]
[0 0 -2 -1]
? algtomatrix(A,[0,x]~,1)
%3 =
[-1 0 0 -1]
[-1 0 1 0]
```

 $\begin{bmatrix} -1 & -1 & 0 & -1 \end{bmatrix}$

Also accepts matrices with coefficients in al.

algtrace(al, x, abs)

Given an element x in al, computes its trace. If al is a table algebra output by algtableinit or if abs = 1, returns the absolute trace of x, which is an element of \mathbb{F}_p or \mathbb{Q} ; if al is the output of alginit and abs = 0 (default), returns the reduced trace of x, which is an element of the center of al.

```
? A = alginit(nfinit(y), [-1,-1]);
? algtrace(A,[5,0,0,1]~)
%2 = 11
? algtrace(A,[5,0,0,1]~,1)
%3 = 22
? nf = nfinit(y^2-5);
? A = alginit(nf,[-1,y]);
? a = [1+x+y,2*y]~*Mod(1,y^2-5)*Mod(1,x^2+1);
? t = algtrace(A,a)
%7 = Mod(2*y + 2, y^2 - 5)
? algtrace(A,a,1)
%8 = 8
? algdegree(A)*nfelttrace(nf,t)
%9 = 8
```

Also accepts a square matrix with coefficients in al.

algtype(al)

Given an algebra *al* output by alginit or by algtableinit, returns an integer indicating the type of algebra:

- 0: not a valid algebra.
- 1: table algebra output by algtableinit.
- 2: central simple algebra output by alginit and represented by a multiplication table over its center.
- 3: central simple algebra output by alginit and represented by a cyclic algebra.

```
? algtype([])
%1 = 0
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? algtype(A)
%4 = 1
? nf = nfinit(y^3-5);
? a = y; b = y^2;
? {m_i = [0,a,0,0;
1,0,0,0;
0,0,0,a;
0,0,0,a;
0,0,1,0];}
? {m_j = [0, 0,b, 0;
0, 0,0,-b;
```

```
1, 0,0, 0;
0,-1,0, 0];}
? {m_k = [0, 0,0,-a*b;
0, 0,b, 0;
0,-a,0, 0;
1, 0,0, 0];}
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf,mt,'x);
? algtype(A)
%12 = 2
? A = alginit(nfinit(y), [-1,-1]);
? algtype(A)
%14 = 3
```

apply(f, A)

Apply the t_CLOSURE f to the entries of A.

- If A is a scalar, return f(A).
- If A is a polynomial or power series $\sum a_i x^i$ (+O(x^N)), apply f on all coefficients and return $\sum f(a_i)x^i$ (+O(x^N)).
- If A is a vector or list $[a_1, ..., a_n]$, return the vector or list $[f(a_1), ..., f(a_n)]$. If A is a matrix, return the matrix whose entries are the f(A[i, j]).

```
? apply(x->x<sup>2</sup>, [1,2,3,4])
%1 = [1, 4, 9, 16]
? apply(x->x<sup>2</sup>, [1,2;3,4])
%2 =
[1 4]
[9 16]
? apply(x->x<sup>2</sup>, 4*x<sup>2</sup> + 3*x+ 2)
%3 = 16*x<sup>2</sup> + 9*x + 4
? apply(sign, 2 - 3* x + 4*x<sup>2</sup> + 0(x<sup>3</sup>))
%4 = 1 - x + x<sup>2</sup> + 0(x<sup>3</sup>)
```

Note that many functions already act componentwise on vectors or matrices, but they almost never act on lists; in this case, apply is a good solution:

```
? L = List([Mod(1,3), Mod(2,4)]);
? lift(L)
*** at top-level: lift(L)
*** ^-----
*** lift: incorrect type in lift.
? apply(lift, L);
%2 = List([1, 2])
```

Remark. For v a t_VEC, t_COL, t_VECSMALL, t_LIST or t_MAT, the alternative set-notations

```
[g(x) | x <- v, f(x)]
[x | x <- v, f(x)]
[g(x) | x <- v]
```

are available as shortcuts for

apply(g, select(f, Vec(v)))
select(f, Vec(v))
apply(g, Vec(v))

respectively:

? L = List([Mod(1,3), Mod(2,4)]); ? [lift(x) | x<-L] %2 = [1, 2]

arg(x, precision)

Argument of the complex number x, such that $-\pi < \arg(x) <= \pi$.

arity(C)

Return the arity of the closure C, i.e., the number of its arguments.

? f1(x,y=0)=x+y; ? arity(f1) %1 = 2 ? f2(t,s[..])=print(t,":",s); ? arity(f2) %2 = 2

Note that a variadic argument, such as s in f2 above, is counted as a single argument.

asin(x, precision)

Principal branch of $\sin^{-1}(x) = -i \log(ix + \sqrt{1 - x^2})$. In particular, $\Re(asin(x)) \in [-\pi/2, \pi/2]$ and if $x \in \mathbb{R}$ and ||x|| > 1 then asin(x) is complex. The branch cut is in two pieces:]-oo, -1], continuous with quadrant II, and [1, +oo] continuous with quadrant IV. The function satisfies iasin(x) = asinh(ix).

asinh(*x*, *precision*)

Principal branch of $\sinh^{-1}(x) = \log(x + \sqrt{1 + x^2})$. In particular $\Im(asinh(x)) \in [-\pi/2, \pi/2]$. The branch cut is in two pieces:] - ioo, -i], continuous with quadrant III and [+i, +ioo], continuous with quadrant I.

asympnum(expr, alpha, precision)

Asymptotic expansion of *expr*, corresponding to a sequence u(n), assuming it has the shape

$$u(n) \sum_{i>=0} a_i n^{-i\alpha}$$

with rational coefficients a_i with reasonable height; the algorithm is heuristic and performs repeated calls to limitnum, with alpha as in limitnum. As in limitnum, u(n) may be given either by a closure n : -- u(n) or as a closure N : -- u(n), the latter being often more efficient.

```
? f(n) = n! / (n^n*exp(-n)*sqrt(n));
? asympnum(f)
%2 = [] \\ failure !
? localprec(57); l = limitnum(f)
%3 = 2.5066282746310005024157652848110452530
? asympnum(n->f(n)/l) \\ normalize
%4 = [1, 1/12, 1/288, -139/51840, -571/2488320, 163879/209018880,
5246819/75246796800]
```

and we indeed get a few terms of Stirling's expansion. Note that it definitely helps to normalize with a limit computed to higher accuracy (as a rule of thumb, multiply the bit accuracy by 1.612):

? 1 = limitnum(f)
? asympnum(n->f(n) / 1) \\ failure again !!!
%6 = []

We treat again the example of the Motzkin numbers M_n given in limitnum:

\\ [M_k, M_{k*2}, ..., M_{k*N}] / (3^n / n^(3/2))
? vM(N, k = 1) =
{ my(q = k*N, V);
 if (q == 1, return ([1/3]));
 V = vector(q); V[1] = V[2] = 1;
 for(n = 2, q - 1,
 V[n+1] = ((2*n + 1)*V[n] + 3*(n - 1)*V[n-1]) / (n + 2));
 f = (n -> 3^n / n^(3/2));
 return (vector(N, n, V[n*k] / f(n*k)));
}
? localprec(100); l = limitnum(n->vM(n,10)); \\ 3/sqrt(12*Pi)
? \p38
? asympnum(n->vM(n,10)/l)
%2 = [1, -3/32, 101/10240, -1617/1638400, 505659/5242880000, ...]

If alpha is not a rational number, loss of accuracy is expected, so it should be precomputed to double accuracy, say:

? \p38 ? asympnum(n->log(1+1/n^Pi),Pi) %1 = [0, 1, -1/2, 1/3, -1/4, 1/5] ? localprec(76); a = Pi; ? asympnum(n->log(1+1/n^Pi), a) \\ more terms %3 = [0, 1, -1/2, 1/3, -1/4, 1/5, -1/6, 1/7, -1/8, 1/9, -1/10, 1/11, -1/12] ? asympnum(n->log(1+1/sqrt(n)),1/2) \\ many more terms %4 = 49

The expression is evaluated for n = 1, 2, ..., N for an N = O(B) if the current bit accuracy is B. If it is not defined for one of these values, translate or rescale accordingly:

```
? asympnum(n->log(1-1/n)) \\ can't evaluate at n = 1 !
   *** at top-level: asympnum(n->log(1-1/n))
   *** ^------
   *** in function asympnum: log(1-1/n)
   *** ^------
   *** log: domain error in log: argument = 0
? asympnum(n->-log(1-1/(2*n)))
%5 = [0, 1/2, 1/8, 1/24, ...]
? asympnum(n->-log(1-1/(n+1)))
%6 = [0, 1, -1/2, 1/3, -1/4, ...]
```

asympnumraw(expr, N, alpha, precision)

Return the N + 1 first terms of asymptotic expansion of *expr*, corresponding to a sequence u(n), as floating point numbers. Assume that the expansion has the shape

$$u(n) \sum_{i>=0} a_i n^{-i\alpha}$$

and return approximation of $[a_0, a_1, ..., a_N]$. The algorithm is heuristic and performs repeated calls to

limitnum, with alpha as in limitnum. As in limitnum, u(n) may be given either by a closure n : --- u(n) or as a closure N : -- v(n), u(n), the latter being often more efficient. This function is related to, but more flexible than, asymptour, which requires rational asymptotic expansions.

```
? f(n) = n! / (n^n*exp(-n)*sqrt(n));
? asympnum(f)
%2 = [] \\ failure !
? v = asympnumraw(f, 10);
? v[1] - sqrt(2*Pi)
%4 = 0.E-37
? bestappr(v / v[1], 2^60)
%5 = [1, 1/12, 1/288, -139/51840, -571/2488320, 163879/209018880,...]
```

and we indeed get a few terms of Stirling's expansion (the first 9 terms are correct). If u(n) has an asymptotic expansion in $n^{-\alpha}$ with α not an integer, the default alpha = 1 is inaccurate:

```
? f(n) = (1+1/n^(7/2))^(n^(7/2));
? v1 = asympnumraw(f,10);
? v1[1] - exp(1)
%8 = 4.62... E-12
? v2 = asympnumraw(f,10,7/2);
? v2[1] - exp(1)
%7 0.E-37
```

As in asymptum, if alpha is not a rational number, loss of accuracy is expected, so it should be precomputed to double accuracy, say.

atan(x, precision)

Principal branch of $tan^{-1}(x) = \log((1+ix)/(1-ix))/2i$. In particular the real part of atan(x) belongs to $] - \pi/2, \pi/2[$. The branch cut is in two pieces:] - ioo, -i[, continuous with quadrant IV, and]i, +ioo[continuous with quadrant II. The function satisfies atan(x) = -iatanh(ix) for all x! = i.

atanh(x, precision)

Principal branch of $tanh^{-1}(x) = \log((1+x)/(1-x))/2$. In particular the imaginary part of atanh(x) belongs to $[-\pi/2, \pi/2]$; if $x \in \mathbb{R}$ and ||x|| > 1 then atanh(x) is complex.

bernfrac(n)

Bernoulli number B_n , where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$,..., expressed as a rational number. The argument *n* should be a nonnegative integer. The function bervec creates a cache of successive Bernoulli numbers which greatly speeds up later calls to bernfrac:

```
? bernfrac(20000);
time = 107 ms.
? bernvec(10000); \\ cache B_0, B_2, ..., B_20000
time = 35,957 ms.
? bernfrac(20000); \\ now instantaneous
?
```

bernpol(n, v)

Bernoulli polynomial B_n in variable v.

? bernpol(1)
%1 = x - 1/2
? bernpol(3)
%2 = x^3 - 3/2*x^2 + 1/2*x

bernreal(n, precision)

Bernoulli number B_n , as bernfrac, but B_n is returned as a real number (with the current precision). The argument n should be a nonnegative integer. The function slows down as the precision increases:

```
? \p1000
? bernreal(200000);
time = 5 ms.
? \p10000
? bernreal(200000);
time = 18 ms.
? \p100000
? bernreal(200000);
time = 84 ms.
```

bernvec(n)

Returns a vector containing, as rational numbers, the Bernoulli numbers B_0, B_2, \ldots, B_{2n} :

? bernvec(5) \\ B_0, B_2..., B_10
%1 = [1, 1/6, -1/30, 1/42, -1/30, 5/66]
? bernfrac(10)
%2 = 5/66

This routine uses a lot of memory but is much faster than repeated calls to bernfrac:

```
? forstep(n = 2, 10000, 2, bernfrac(n))
time = 18,245 ms.
? bernvec(5000);
time = 1,338 ms.
```

The computed Bernoulli numbers are stored in an incremental cache which makes later calls to bernfrac and bernreal instantaneous in the cache range: re-running the same previous bernfrac s after the bernvec call gives:

```
? forstep(n = 2, 10000, 2, bernfrac(n))
time = 1 ms.
```

The time and space complexity of this function are $O(n^2)$; in the feasible range $n \le 10^5$ (requires about two hours), the practical time complexity is closer to $O(n^{\log_2 6})$.

besselh1(nu, x, precision)

 H^1 -Bessel function of index *nu* and argument *x*.

besselh2(nu, x, precision)

 H^2 -Bessel function of index *nu* and argument *x*.

besseli(nu, x, precision)

I-Bessel function of index *nu* and argument *x*. If *x* converts to a power series, the initial factor $(x/2)^{\nu}/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

besselj(*nu*, *x*, *precision*)

J-Bessel function of index *nu* and argument *x*. If *x* converts to a power series, the initial factor $(x/2)^{\nu}/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

besseljh(n, x, precision)

J-Bessel function of half integral index. More precisely, bessel jh(n,x) computes $J_{n+1/2}(x)$ where n

must be of type integer, and x is any element of \mathbb{C} . In the present version **2.13.3**, this function is not very accurate when x is small.

besselk(nu, x, precision)

K-Bessel function of index nu and argument x.

```
besseln(nu, x, precision)
```

Deprecated alias for bessely.

```
bessely(nu, x, precision)
```

Y-Bessel function of index nu and argument x.

bestappr(x, B)

Using variants of the extended Euclidean algorithm, returns a rational approximation a/b to x, whose denominator is limited by B, if present. If B is omitted, returns the best approximation affordable given the input accuracy; if you are looking for true rational numbers, presumably approximated to sufficient accuracy, you should first try that option. Otherwise, B must be a positive real scalar (impose 0 < b <= B).

• If x is a t_REAL or a t_FRAC, this function uses continued fractions.

```
? bestappr(Pi, 100)
%1 = 22/7
? bestappr(0.1428571428571428571428571428571429)
%2 = 1/7
? bestappr([Pi, sqrt(2) + 'x], 10^3)
%3 = [355/113, x + 1393/985]
```

By definition, a/b is the best rational approximation to x if ||bx - a|| < ||vx - u|| for all integers (u, v) with $0 < v \le B$. (Which implies that n/d is a convergent of the continued fraction of x.)

If x is a t_INTMOD modulo N or a t_PADIC of precision N = p^k, this function performs rational modular reconstruction modulo N. The routine then returns the unique rational number a/b in coprime integers ||a|| < N/2B and b <= B which is congruent to x modulo N. Omitting B amounts to choosing it of the order of √N/2. If rational reconstruction is not possible (no suitable a/b exists), returns [].

```
? bestappr(Mod(18526731858, 11^10))
%1 = 1/7
? bestappr(Mod(18526731858, 11^20))
%2 = []
? bestappr(3 + 5 + 3*5^2 + 5^3 + 3*5^4 + 5^5 + 3*5^6 + 0(5^7))
%2 = -1/3
```

In most concrete uses, B is a prime power and we performed Hensel lifting to obtain x.

The function applies recursively to components of complex objects (polynomials, vectors,...). If rational reconstruction fails for even a single entry, returns [].

bestapprPade(x, B)

Using variants of the extended Euclidean algorithm (Padé approximants), returns a rational function approximation a/b to x, whose denominator is limited by B, if present. If B is omitted, return the best approximation affordable given the input accuracy; if you are looking for true rational functions, presumably approximated to sufficient accuracy, you should first try that option. Otherwise, B must be a nonnegative real (impose $0 \le degree(b) \le B$).

• If x is a t_POLMOD modulo N this function performs rational modular reconstruction modulo N. The routine then returns the unique rational function a/b in coprime polynomials, with $degree(b) \le B$ and degree(a) minimal, which is congruent to x modulo N. Omitting B amounts to choosing it equal

to the floor of degree(N)/2. If rational reconstruction is not possible (no suitable a/b exists), returns [].

```
? T = Mod(x^3 + x^2 + x + 3, x^4 - 2);

? bestapprPade(T)

%2 = (2^*x - 1)/(x - 1)

? U = Mod(1 + x + x^2 + x^3 + x^5, x^9);

? bestapprPade(U) \\ internally chooses B = 4

%3 = []

? bestapprPade(U, 5) \\ with B = 5, a solution exists

%4 = (2^*x^4 + x^3 - x - 1)/(-x^5 + x^3 + x^2 - 1)
```

• If x is a t_SER, we implicitly convert the input to a t_POLMOD modulo $N = t^k$ where k is the series absolute precision.

```
? T = 1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + 0(t^7); \ \mod t^7
? bestapprPade(T)
%1 = 1/(-t + 1)
```

• If x is a t_RFRAC, we implicitly convert the input to a t_POLMOD modulo $N = t^k$ where k = 2B + 1. If B was omitted, we return x:

```
? T = (4*t<sup>2</sup> + 2*t + 3)/(t+1)<sup>10</sup>;
? bestapprPade(T,1)
%2 = [] \\ impossible
? bestapprPade(T,2)
%3 = 27/(337*t<sup>2</sup> + 84*t + 9)
? bestapprPade(T,3)
%4 = (4253*t - 3345)/(-39007*t<sup>3</sup> - 28519*t<sup>2</sup> - 8989*t - 1115)
```

The function applies recursively to components of complex objects (polynomials, vectors,...). If rational reconstruction fails for even a single entry, return [].

bestapprnf(*V*, *T*, rootT, precision)

T being an integral polynomial and *V* being a scalar, vector, or matrix with complex coefficients, return a reasonable approximation of *V* with polmods modulo *T*. *T* can also be any number field structure, in which case the minimal polynomial attached to the structure (:math:`T`.pol) is used. The *rootT* argument, if present, must be an element of polroots(:math:`T)` (or :math:`T`.pol), i.e. a complex root of *T* fixing an embedding of $\mathbb{Q}[x]/(T)$ into \mathbb{C} .

```
? bestapprnf(sqrt(5), polcyclo(5))
%1 = Mod(-2*x^3 - 2*x^2 - 1, x^4 + x^3 + x^2 + x + 1)
? bestapprnf(sqrt(5), polcyclo(5), exp(4*I*Pi/5))
%2 = Mod(2*x^3 + 2*x^2 + 1, x^4 + x^3 + x^2 + x + 1)
```

When the output has huge rational coefficients, try to increase the working realbitprecision: if the answer does not stabilize, consider that the reconstruction failed. Beware that if T is not Galois over \mathbb{Q} , some embeddings may not allow to reconstruct V:

```
? T = x^3-2; vT = polroots(T); z = 3*2^(1/3)+1;
? bestapprnf(z, T, vT[1])
%2 = Mod(3*x + 1, x^3 - 2)
```

```
? bestapprnf(z, T, vT[2])
%3 = 4213714286230872/186454048314072 \\ close to 3*2^(1/3) + 1
```

bezout(x, y)

Deprecated alias for gcdext

bezoutres(A, B, v)

Deprecated alias for polresultantext

bigomega(x)

Number of prime divisors of the integer ||x|| counted with multiplicity:

```
? factor(392)
%1 =
[2 3]
[7 2]
? bigomega(392)
%2 = 5; \\ = 3+2
? omega(392)
%3 = 2; \\ without multiplicity
```

binary(x)

Outputs the vector of the binary digits of ||x||. Here x can be an integer, a real number (in which case the result has two components, one for the integer part, one for the fractional part) or a vector/matrix.

```
? binary(10)
%1 = [1, 0, 1, 0]
? binary(3.14)
%2 = [[1, 1], [0, 0, 1, 0, 0, 0, [...]]
? binary([1,2])
%3 = [[1], [1, 0]]
```

For integer $x \ge 1$, the number of bits is logint(x, 2) + 1. By convention, 0 has no digits:

? binary(0) %4 = []

binomial(x, k)

binomial coefficient binom xk. Here k must be an integer, but x can be any PARI object.

```
? binomial(4,2)
%1 = 6
? n = 4; vector(n+1, k, binomial(n,k-1))
%2 = [1, 4, 6, 4, 1]
```

The argument k may be omitted if x = n is a nonnegative integer; in this case, return the vector with n + 1 components whose k + 1-th entry is binomial(n, k)

? binomial(4) %3 = [1, 4, 6, 4, 1]

bitand(x, y)

Bitwise and of two integers x and y, that is the integer

 $\sum_{i} (x_i \text{ and } y_i) 2^i$

Negative numbers behave 2-adically, i.e. the result is the 2-adic limit of $bitand(x_n, y_n)$, where x_n and y_n are nonnegative integers tending to x and y respectively. (The result is an ordinary integer, possibly negative.)

? bitand(5, 3)
%1 = 1
? bitand(-5, 3)
%2 = 3
? bitand(-5, -3)
%3 = -7

bitneg(x, n)

bitwise negation of an integer x, truncated to n bits, $n \ge 0$, that is the integer

$$\sum_{i=0}^{n-1} not(x_i) 2^i.$$

The special case n = -1 means no truncation: an infinite sequence of leading 1 is then represented as a negative number.

See bitand (in the PARI manual) for the behavior for negative arguments.

bitnegimply(x, y)

Bitwise negated imply of two integers x and y (or not (x = y)), that is the integer

 $\sum (x_i \ and not(y_i)) 2^i$

See bitand (in the PARI manual) for the behavior for negative arguments.

bitor(x, y)

bitwise (inclusive) or of two integers x and y, that is the integer

$$\sum (x_i \text{ or } y_i)2^i$$

See bitand (in the PARI manual) for the behavior for negative arguments.

bitprecision(x, n)

The function behaves differently according to whether n is present or not. If n is missing, the function returns the (floating point) precision in bits of the PARI object x.

If x is an exact object, the function returns +00.

```
? bitprecision(exp(1e-100))
%1 = 512 \\ 512 bits
? bitprecision( [ exp(1e-100), 0.5 ] )
%2 = 128 \\ minimal accuracy among components
? bitprecision(2 + x)
%3 = +oo \\ exact object
```

Use getlocalbitprec() to retrieve the working bit precision (as modified by possible localbitprec statements).

If n is present and positive, the function creates a new object equal to x with the new bit-precision roughly n. In fact, the smallest multiple of 64 (resp. 32 on a 32-bit machine) larger than or equal to n.

For x a vector or a matrix, the operation is done componentwise; for series and polynomials, the operation is done coefficientwise. For real x, n is the number of desired significant *bits*. If n is smaller than the precision of x, x is truncated, otherwise x is extended with zeros. For exact or non-floating-point types, no change.

```
? bitprecision(Pi, 10) \\ actually 64 bits ~ 19 decimal digits
%1 = 3.141592653589793239
? bitprecision(1, 10)
%2 = 1
? bitprecision(1 + 0(x), 10)
%3 = 1 + 0(x)
? bitprecision(2 + 0(3^5), 10)
%4 = 2 + 0(3^5)
```

bittest(x, n)

Outputs the n - th bit of x starting from the right (i.e. the coefficient of 2^n in the binary expansion of x). The result is 0 or 1. For $x \ge 1$, the highest 1-bit is at n = logint(x) (and bigger n gives 0).

```
? bittest(7, 0)
%1 = 1 \\ the bit 0 is 1
? bittest(7, 2)
%2 = 1 \\ the bit 2 is 1
? bittest(7, 3)
%3 = 0 \\ the bit 3 is 0
```

See bitand (in the PARI manual) for the behavior at negative arguments.

bitxor(x, y)

Bitwise (exclusive) or of two integers x and y, that is the integer

 $\sum (x_i \ xor \ y_i) 2^i$

See bitand (in the PARI manual) for the behavior for negative arguments.

bnfcertify(bnf, flag)

bnf being as output by bnfinit, checks whether the result is correct, i.e. whether it is possible to remove the assumption of the Generalized Riemann Hypothesis. It is correct if and only if the answer is 1. If it is incorrect, the program may output some error message, or loop indefinitely. You can check its progress by increasing the debug level. The *bnf* structure must contain the fundamental units:

```
? K = bnfinit(x^3+2^2^3+1); bnfcertify(K)
  *** at top-level: K=bnfinit(x^3+2^2^3+1); bnfcertify(K)
  *** ^------
  *** bnfcertify: precision too low in makeunits [use bnfinit(,1)].
? K = bnfinit(x^3+2^2^3+1, 1); \\ include units
? bnfcertify(K)
%3 = 1
```

If flag is present, only certify that the class group is a quotient of the one computed in bnf (much simpler in general); likewise, the computed units may form a subgroup of the full unit group. In this variant, the units are no longer needed:

? K = bnfinit(x^3+2^2*3+1); bnfcertify(K, 1)
%4 = 1

bnfdecodemodule(nf, m)

If m is a module as output in the first component of an extension given by bnrdisclist, outputs the true module.

```
? K = bnfinit(x^2+23); L = bnrdisclist(K, 10); s = L[2]
%1 = [[[Vecsmall([8]), Vecsmall([1])], [[0, 0, 0]]],
  [[Vecsmall([9]), Vecsmall([1])], [[0, 0, 0]]]]
? bnfdecodemodule(K, s[1][1])
%2 =
[2 0]
[0 1]
? bnfdecodemodule(K,s[2][1])
%3 =
[2 1]
[0 1]
```

bnfinit(P, flag, tech, precision)

Initializes a bnf structure. Used in programs such as bnfisprincipal, bnfisunit or bnfnarrow. By default, the results are conditional on the GRH, see GRHbnf (in the PARI manual). The result is a 10-component vector *bnf*.

This implements Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible polynomial P with integer coefficients. The meaning of *flag* is as follows:

• *flag* = 0 (default). This is the historical behavior, kept for compatibility reasons and speed. It has severe drawbacks but is likely to be a little faster than the alternative, twice faster say, so only use it if speed is paramount, you obtain a useful speed gain for the fields under consideration, and you are only interested in the field invariants such as the classgroup structure or its regulator. The computations involve exact algebraic numbers which are replaced by floating point embeddings for the sake of speed. If the precision is insufficient, gp may not be able to compute fundamental units, nor to solve some discrete logarithm problems. It *may* be possible to increase the precision of the bnf structure using nfnewprec but this may fail, in particular when fundamental units are large. In short, the resulting bnf structure is correct and contains useful information but later function calls to bnfisprincpal or bnrclassfield may fail.

When flag = 1, we keep an exact algebraic version of all floating point data and this allows to guarantee that functions using the structure will always succeed, as well as to compute the fundamental units exactly. The units are computed in compact form, as a product of small S-units, possibly with huge exponents. This flag also allows bnfisprincipal to compute generators of principal ideals in factored form as well. Be warned that expanding such products explicitly can take a very long time, but they can easily be mapped to floating point or ℓ -adic embeddings of bounded accuracy, or to $K^*/(K^*)^{\ell}$, and this is enough for applications. In short, this flag should be used by default, unless you have a very good reason for it, for instance building massive tables of class numbers, and you do not care about units or the effect large units would have on your computation.

tech is a technical vector (empty by default, see GRHbnf (in the PARI manual)). Careful use of this parameter may speed up your computations, but it is mostly obsolete and you should leave it alone.

The components of a *bnf* are technical. In fact: *never access a component directly, always use a proper member function.* However, for the sake of completeness and internal documentation, their description is

as follows. We use the notations explained in the book by H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Maths **138**, Springer-Verlag, 1993, Section 6.5, and subsection 6.5.5 in particular.

bnf[1] contains the matrix W, i.e. the matrix in Hermite normal form giving relations for the class group on prime ideal generators $(p_i)_{1 \le i \le r}$.

bnf[2] contains the matrix B, i.e. the matrix containing the expressions of the prime ideal factorbase in terms of the p_i . It is an rxc matrix.

bnf[3] contains the complex logarithmic embeddings of the system of fundamental units which has been found. It is an $(r_1 + r_2)x(r_1 + r_2 - 1)$ matrix.

bnf[4] contains the matrix M''_C of Archimedean components of the relations of the matrix (W||B).

bnf[5] contains the prime factor base, i.e. the list of prime ideals used in finding the relations.

bnf[6] contains a dummy 0.

bnf[7] or :emphasis: bnf.nf is equal to the number field data nf as would be given by nfinit.

bnf[8] is a vector containing the classgroup :emphasis: `bnf.clgp` as a finite abelian group, the regulator :emphasis: `bnf.reg`, the number of roots of unity and a generator :emphasis: `bnf.tu`, the fundamental units *in expanded form* :emphasis: `bnf.fu`. If the fundamental units were omitted in the *bnf*, :emphasis: `bnf.fu` returns the sentinel value 0. If flag = 1, this vector contain also algebraic data corresponding to the fundamental units and to the discrete logarithm problem (see bnfisprincipal). In particular, if flag = 1 we may *only* know the units in factored form: the first call to :emphasis: `bnf.fu` expands them, which may be very costly, then caches the result.

bnf[9] is a vector used in bnfisprincipal only and obtained as follows. Let D = UWV obtained by applying the Smith normal form algorithm to the matrix W (= bnf[1]) and let U_r be the reduction of Umodulo D. The first elements of the factorbase are given (in terms of bnf.gen) by the columns of U_r , with Archimedean component g_a ; let also GD_a be the Archimedean components of the generators of the (principal) ideals defined by the bnf.gen[i]^bnf.cyc[i]. Then $bnf[9] = [U_r, g_a, GD_a]$, followed by technical exact components which allow to recompute g_a and GD_a to higher accuracy.

bnf[10] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available, which is rarely needed, hence would be too expensive to compute during the initial bnfinit call. For instance, the generators of the principal ideals bnf.gen[i]^bnf.cyc[i] (during a call to bnrisprincipal), or those corresponding to the relations in W and B (when the bnf internal precision needs to be increased).

bnfisintnorm(bnf, x)

Computes a complete system of solutions (modulo units of positive norm) of the absolute norm equation Norm(a) = x, where a is an integer in bnf. If bnf has not been certified, the correctness of the result depends on the validity of GRH.

See also bnfisnorm.

bnfisnorm(bnf, x, flag)

Tries to tell whether the rational number x is the norm of some element y in bnf. Returns a vector [a, b] where x = Norm(a) * b. Looks for a solution which is an S-unit, with S a certain set of prime ideals containing (among others) all primes dividing x. If bnf is known to be Galois, you may set flag = 0 (in this case, x is a norm iff b = 1). If flag is nonzero the program adds to S the following prime ideals, depending on the sign of flag. If flag > 0, the ideals of norm less than flag. And if flag < 0 the ideals dividing flag.

Assuming GRH, the answer is guaranteed (i.e. x is a norm iff b = 1), if S contains all primes less than $12 \log(\operatorname{disc}(Bnf))^2$, where Bnf is the Galois closure of bnf.

See also bnfisintnorm.

bnfisprincipal(bnf, x, flag)

bnf being the number field data output by bnfinit, and x being an ideal, this function tests whether the ideal is principal or not. The result is more complete than a simple true/false answer and solves a general discrete logarithm problem. Assume the class group is $\oplus (\mathbb{Z}/d_i\mathbb{Z})g_i$ (where the generators g_i and their orders d_i are respectively given by bnf.gen and bnf.cyc). The routine returns a row vector [e, t], where e is a vector of exponents $0 \le e_i < d_i$, and t is a number field element such that

$$x = (t) \prod_{i} g_i^{e_i}.$$

For given g_i (i.e. for a given bnf), the e_i are unique, and t is unique modulo units.

In particular, x is principal if and only if e is the zero vector. Note that the empty vector, which is returned when the class number is 1, is considered to be a zero vector (of dimension 0).

```
? K = bnfinit(y^2+23);
? K.cyc
%2 = [3]
? K.gen
%3 = [[2, 0; 0, 1]] \\ a prime ideal above 2
? P = idealprimedec(K,3)[1]; \\ a prime ideal above 3
? v = bnfisprincipal(K, P)
%5 = [[2]~, [3/4, 1/4]~]
? idealmul(K, v[2], idealfactorback(K, K.gen, v[1]))
%6 =
[3 0]
[0 1]
? % == idealhnf(K, P)
%7 = 1
```

The binary digits of *flag* mean:

• 1: If set, outputs [e, t] as explained above, otherwise returns only e, which is much easier to compute. The following idiom only tests whether an ideal is principal:

is_principal(bnf, x) = !bnfisprincipal(bnf,x,0);

- 2: It may not be possible to recover t, given the initial accuracy to which the bnf structure was computed. In that case, a warning is printed and t is set equal to the empty vector []~. If this bit is set, increase the precision and recompute needed quantities until t can be computed. Warning: setting this may induce *lengthy* computations and you should consider using flag 4 instead.
- 4: Return t in factored form (compact representation), as a small product of S-units for a small set of finite places S, possibly with huge exponents. This kind of result can be cheaply mapped to K*/(K*)^ℓ or to C or Q_p to bounded accuracy and this is usually enough for applications. Explicitly expanding such a compact representation is possible using nffactorback but may be very costly. The algorithm is guaranteed to succeed if the bnf was computed using bnfinit(,1). If not, the algorithm may fail to compute a huge generator in this case (and replace it by []~). This is orders of magnitude faster than flag 2 when the generators are indeed large.

bnfissunit(bnf, sfu, x)

This function is obsolete, use bnfisunit.

bnfisunit(bnf, x, U)

bnf being the number field data output by bnfinit and x being an algebraic number (type integer, rational or polmod), this outputs the decomposition of x on the fundamental units and the roots of unity if x is a unit, the empty vector otherwise. More precisely, if $u_1, \ldots, :math: u_r$ are the fundamental units, and ζ is the generator of the group of roots of unity (bnf.tu), the output is a vector $[x_1, \ldots, x_r, x_{r+1}]$ such that $x = u_1^{x_1} \ldots u_r^{x_r} . \zeta^{x_{r+1}}$. The x_i are integers but the last one (i = r + 1) is only defined modulo the order w of ζ and is guaranteed to be in [0, w].

Note that *bnf* need not contain the fundamental units explicitly: it may contain the placeholder 0 instead:

```
? setrand(1); bnf = bnfinit(x^2-x-100000);
? bnf.fu
%2 = 0
? u = [119836165644250789990462835950022871665178127611316131167, \
379554884019013781006303254896369154068336082609238336]~;
? bnfisunit(bnf, u)
%3 = [-1, 0]~
```

The given u is $1/u_1$, where u_1 is the fundamental unit implicitly stored in *bnf*. In this case, u_1 was not computed and stored in algebraic form since the default accuracy was too low. Re-run the **bnfinit** command at \g1 or higher to see such diagnostics.

This function allows x to be given in factored form, but it then assumes that x is an actual unit. (Because it is general too costly to check whether this is the case.)

```
? { v = [2, 85; 5, -71; 13, -162; 17, -76; 23, -37; 29, -104; [224, 1]~, -66;
[-86, 1]~, 86; [-241, 1]~, -20; [44, 1]~, 30; [124, 1]~, 11; [125, -1]~, -11;
[-214, 1]~, 33; [-213, -1]~, -33; [189, 1]~, 74; [190, -1]~, 104;
[-168, 1]~, 2; [-167, -1]~, -8]; }
? bnfisunit(bnf,v)
%5 = [1, 0]~
```

Note that v is the fundamental unit of **bnf** given in compact (factored) form.

If the argument U is present, as output by bnfunits(bnf, S), then the function decomposes x on the S-units generators given in U[1].

```
? bnf = bnfinit(x^4 - x^3 + 4*x^2 + 3*x + 9, 1);
? bnf.sign
\%2 = [0, 2]
? S = idealprimedec(bnf,5); #S
\%3 = 2
? US = bnfunits(bnf,S);
? g = US[1]; #g \\ #S = #g, four S-units generators, in factored form
%5 = 4
? g[1]
\%6 = [[6, -3, -2, -2] \sim 1]
? g[2]
%7 =
[[-1, 1/2, -1/2, -1/2] \sim 1]
[[4, -2, -1, -1] \sim 1]
? [nffactorback(bnf, x) | x <- g]
%8 = [[6, -3, -2, -2]~, [-5, 5, 0, 0]~, [-1, 1, -1, 0]~,
[1, -1, 0, 0]~]
```

```
? u = [10,-40,24,11]~;
? a = bnfisunit(bnf, u, US)
%9 = [2, 0, 1, 4]~
? nffactorback(bnf, g, a) \\ prod_i g[i]^a[i] still in factored form
%10 =
[[6, -3, -2, -2]~ 2]
[ [0, 0, -1, -1]~ 1]
[ [2, -1, -1, 0]~ -2]
[ [1, 1, 0, 0]~ 2]
[ [1, 1, 0, 0]~ 2]
[ [-1, 1, 1, 1]~ -1]
[ [1, -1, 0, 0]~ 4]
? nffactorback(bnf,%) \\ u = prod_i g[i]^a[i]
%11 = [10, -40, 24, 11]~
```

bnflog(bnf, l)

Let *bnf* be a *bnf* structure attached to the number field F and let l be a prime number (hereafter denoted ℓ for typographical reasons). Return the logarithmic ℓ -class group Cl_F of F. This is an abelian group, conjecturally finite (known to be finite if F/\mathbb{Q} is abelian). The function returns if and only if the group is indeed finite (otherwise it would run into an infinite loop). Let $S = p_1, ..., p_k$ be the set of ℓ -adic places (maximal ideals containing ℓ). The function returns $[D, G(\ell), G']$, where

- D is the vector of elementary divisors for Cl_F .
- $G(\ell)$ is the vector of elementary divisors for the (conjecturally finite) abelian group

where the :math: p_i 'are the : math : ' ℓ '-adicplaces of : math : 'F'; this is a subgroup of : math : 'Cl'.

• G' is the vector of elementary divisors for the ℓ -Sylow Cl' of the S-class group of F; the group Cl maps to Cl' with a simple co-kernel.

bnflogdegree(nf, A, l)

Let *nf* be a *nf* structure attached to a number field *F*, and let *l* be a prime number (hereafter denoted ℓ). The ℓ -adified group of id\`{e}les of *F* quotiented by the group of logarithmic units is identified to the ℓ -group of logarithmic divisors $\oplus \mathbb{Z}_{\ell}[p]$, generated by the maximal ideals of *F*.

The degree map \deg_F is additive with values in \mathbb{Z}_ℓ , defined by $\deg_F p = f_p \deg_\ell p$, where the integer f_p is as in **bnflogef** and $\deg_\ell p$ is $\log_\ell p$ for $p! = \ell$, $\log_\ell (1 + \ell)$ for $p = \ell! = 2$ and $\log_\ell (1 + 2^2)$ for $p = \ell = 2$.

Let $A = \prod p^{n_p}$ be an ideal and let $A = \sum n_p[p]$ be the attached logarithmic divisor. Return the exponential of the ℓ -adic logarithmic degree deg_F A, which is a natural number.

bnflogef(nf, pr)

Let *nf* be a *nf* structure attached to a number field *F* and let *pr* be a *prid* structure attached to a maximal ideal p/p. Return $[e(F_p/\mathbb{Q}_p), f(F_p/\mathbb{Q}_p)]$ the logarithmic ramification and residue degrees. Let $\mathbb{Q}_p^c/\mathbb{Q}_p$ be the cyclotomic \mathbb{Z}_p -extension, then $e = [F_p : F_p \cap \mathbb{Q}_p^c]$ and $f = [F_p \cap \mathbb{Q}_p^c : \mathbb{Q}_p]$. Note that e f = e(p/p)f(p/p), where e(p/p) and f(p/p) denote the usual ramification and residue degrees.

```
? F = nfinit(y^6 - 3*y^5 + 5*y^3 - 3*y + 1);
? bnflogef(F, idealprimedec(F,2)[1])
%2 = [6, 1]
? bnflogef(F, idealprimedec(F,5)[1])
%3 = [1, 2]
```

bnfnarrow(bnf)

bnf being as output by bnfinit, computes the narrow class group of *bnf*. The output is a 3-component row vector v analogous to the corresponding class group component :emphasis: `bnf.clgp`: the first component is the narrow class number :math: `v.no`, the second component is a vector containing the SNF cyclic components :math: `v.cyc` of the narrow class group, and the third is a vector giving the generators of the corresponding :math: `v.gen` cyclic groups. Note that this function is a special case of bnrinit; the *bnf* need not contain fundamental units.

bnfsignunit(bnf)

bnf being as output by bnfinit, this computes an $r_1x(r_1 + r_2 - 1)$ matrix having 1 components, giving the signs of the real embeddings of the fundamental units. The following functions compute generators for the totally positive units:

```
/* exponents of totally positive units generators on K.tu, K.fu */
tpuexpo(K)=
{ my(M, S = bnfsignunit(K), [m,n] = matsize(S));
   \\ m = K.r1, n = r1+r2-1
   S = matrix(m,n, i,j, if (S[i,j] < 0, 1,0));
   S = concat(vectorv(m,i,1), S); \\ add sign(-1)
   M = matkermod(S, 2);
   if (M, mathnfmodid(M, 2), 2*matid(n+1))
}
/* totally positive fundamental units of bnf K */
tpu(K)=
{ my(ex = tpuexpo(K)[,^1]); \\ remove ex[,1], corresponds to 1 or -1
   my(v = concat(K.tu[2], K.fu));
   [ nffactorback(K, v, c) | c <- ex];
}</pre>
```

bnfsunit(bnf, S, precision)

Computes the fundamental S-units of the number field bnf (output by bnfinit), where S is a list of prime ideals (output by idealprimedec). The output is a vector v with 6 components.

v[1] gives a minimal system of (integral) generators of the S-unit group modulo the unit group.

- v[2] contains technical data needed by bnfissunit.
- v[3] is an obsoleted component, now the empty vector.

v[4] is the S-regulator (this is the product of the regulator, the S-class number and the natural logarithms of the norms of the ideals in S).

v[5] gives the S-class group structure, in the usual abelian group format: a vector whose three components give in order the S-class number, the cyclic components and the generators.

v[6] is a copy of S.

bnfunits(bnf, S)

Return the fundamental units of the number field bnf output by bnfinit; if S is present and is a list of prime ideals, compute fundamental S-units instead. The first component of the result contains independent integral S-units generators: first nonunits, then $r_1 + r_2 - 1$ fundamental units, then the torsion unit. The result may be used as an optional argument to bnfisunit. The units are given in compact form: no expensive computation is attempted if the *bnf* does not already contain units.

```
? bnf = bnfinit(x^4 - x^3 + 4*x^2 + 3*x + 9, 1);
? bnf.sign \ r1 + r2 - 1 = 1
\%2 = [0, 2]
? U = bnfunits(bnf); u = U[1];
? #u \ r1 + r2 = 2 units
\%5 = 2:
? u[1] \\ fundamental unit as factorization matrix
%6 =
[[0, 0, -1, -1] \sim 1]
[[2, -1, -1, 0] \sim -2]
[ [1, 1, 0, 0] \sim 2 ]
[ [-1, 1, 1, 1]~ -1]
? u[2] \\ torsion unit as factorization matrix
%7 =
[[1, -1, 0, 0] \sim 1]
? [nffactorback(bnf, z) | z <- u] \setminus same units in expanded form
\%8 = [[-1, 1, -1, 0] \sim, [1, -1, 0, 0] \sim]
```

Now an example involving S-units for a nontrivial S:

```
? S = idealprimedec(bnf,5); #S
\%9 = 2
? US = bnfunits(bnf, S); uS = US[1];
? g = [nffactorback(bnf, z) | z <- uS] \\ now 4 units</pre>
11 = [[6, -3, -2, -2]], [-5, 5, 0, 0], [-1, 1, -1, 0], [1, -1, 0, 0]]
? bnfisunit(bnf,[10,-40,24,11]~)
\%12 = [] \sim \setminus  not a unit
? e = bnfisunit(bnf, [10,-40,24,11]~, US)
%13 = [2, 0, 1, 4] \sim \setminus \ldots but an S-unit
? nffactorback(bnf, g, e)
%14 = [10, -40, 24, 11]~
? nffactorback(bnf, uS, e) \\ in factored form
%15 =
[[6, -3, -2, -2] \sim 2]
[ [0, 0, -1, -1] \sim 1 ]
[ [2, -1, -1, 0]~ -2]
[ [1, 1, 0, 0] \sim 2 ]
[ [-1, 1, 1, 1]~ -1]
[ [1, -1, 0, 0]~ 4]
```

Note that in more complicated cases, any nffactorback fully expanding an element in factored form could be *very* expensive. On the other hand, the final example expands a factorization whose components are themselves in factored form, hence the result is a factored form: this is a cheap operation.

bnrL1(bnr, H, flag, precision)

Let *bnr* be the number field data output by **bnrinit** and *H* be a square matrix defining a congruence subgroup of the ray class group corresponding to *bnr* (the trivial congruence subgroup if omitted). This function returns, for each character χ of the ray class group which is trivial on *H*, the value at s = 1 (or s = 0) of the abelian *L*-function attached to χ . For the value at s = 0, the function returns in fact for each χ a vector $[r_{\chi}, c_{\chi}]$ where

$$L(s,\chi) = c.s^r + O(s^{r+1})$$

near 0.

The argument *flag* is optional, its binary digits mean 1: compute at s = 0 if unset or s = 1 if set, 2: compute the primitive *L*-function attached to χ if unset or the *L*-function with Euler factors at prime ideals dividing the modulus of *bnr* removed if set (that is $L_S(s, \chi)$, where *S* is the set of infinite places of the number field together with the finite prime ideals dividing the modulus of *bnr*), 3: return also the character if set.

```
K = bnfinit(x^2-229);
bnr = bnrinit(K,1);
bnrL1(bnr)
```

returns the order and the first nonzero term of $L(s, \chi)$ at s = 0 where χ runs through the characters of the class group of $K = \mathbb{Q}(\sqrt{229})$. Then

```
bnr2 = bnrinit(K,2);
bnrL1(bnr2,,2)
```

returns the order and the first nonzero terms of $L_S(s, \chi)$ at s = 0 where χ runs through the characters of the class group of K and S is the set of infinite places of K together with the finite prime 2. Note that the ray class group modulo 2 is in fact the class group, so bnrL1(bnr2,0) returns the same answer as bnrL1(bnr,0).

This function will fail with the message

*** bnrL1: overflow in zeta_get_N0 [need too many primes].

if the approximate functional equation requires us to sum too many terms (if the discriminant of K is too large).

bnrchar(bnr, g, v)

Returns all characters χ on bnr.clgp such that $\chi(g_i) = e(v_i)$, where $e(x) = \exp(2i\pi x)$. If v is omitted, returns all characters that are trivial on the g_i . Else the vectors g and v must have the same length, the g_i must be ideals in any form, and each v_i is a rational number whose denominator must divide the order of g_i in the ray class group. For convenience, the vector of the g_i can be replaced by a matrix whose columns give their discrete logarithm, as given by bnrisprincipal; this allows to specify abstractly a subgroup of the ray class group.

? bnr = bnrinit(bnfinit(x), [160,[1]], 1); /* (Z/160Z)^* */ ? bnr.cyc %2 = [8, 4, 2] ? g = bnr.gen; ? bnrchar(bnr, g, [1/2,0,0]) %4 = [[4, 0, 0]] \\ a unique character ? bnrchar(bnr, [g[1],g[3]]) \\ all characters trivial on g[1] and g[3] %5 = [[0, 1, 0], [0, 2, 0], [0, 3, 0], [0, 0, 0]] ? bnrchar(bnr, [1,0,0;0,1,0;0,0,2]) %6 = [[0, 0, 1], [0, 0, 0]] \\ characters trivial on given subgroup

bnrclassfield(bnr, subgp, flag, precision)

bnr being as output by bnrinit, returns a relative equation for the class field corresponding to the congruence group defined by (bnr, subgp) (the full ray class field if *subgp* is omitted). The subgroup can also be a t_INT *n*, meaning $n.Cl_f$. The function also handles a vector of subgroup, e.g, from subgrouplist and returns the vector of individual results in this case.

If flag = 0, returns a vector of polynomials such that the compositum of the corresponding fields is the class field; if flag = 1 returns a single polynomial; if flag = 2 returns a single absolute polynomial.

```
? bnf = bnfinit(y^3+14*y-1); bnf.cyc
\%1 = [4, 2]
? pol = bnrclassfield(bnf,,1) \\ Hilbert class field
2 = x^8 - 2x^7 + \ldots + Mod(11x^2 - 82x^2 + 116, y^3 + 14x^2 - 1)
? rnfdisc(bnf,pol)[1]
\%3 = 1
? bnr = bnrinit(bnf,3*5*7); bnr.cyc
\%4 = [24, 12, 12, 2]
? bnrclassfield(bnr,2) \\ maximal 2-elementary subextension
\%5 = [x^2 + (-21^*y - 10^5), x^2 + (-5^*y - 2^5), x^2 + (-y - 5), x^2 + (-y - 1)]
\\ quadratic extensions of maximal conductor
? bnrclassfield(bnr, subgrouplist(bnr,[2]))
\%6 = [[x^2 - 105], [x^2 + (-105*y^2 - 1260)], [x^2 + (-105*y - 525)],
[x^2 + (-105*y - 105)]]
? #bnrclassfield(bnr,subgrouplist(bnr,[2],1)) \\ all quadratic extensions
%7 = 15
```

When the subgroup contains nCl_f , where n is fixed, it is advised to directly compute the bnr modulo n to avoid expensive discrete logarithms:

```
? bnf = bnfinit(y^2-5); p = 1594287814679644276013;
? bnr = bnrinit(bnf,p); \\ very slow
time = 24,146 ms.
? bnrclassfield(bnr, 2) \\ ... even though the result is trivial
%3 = [x^2 - 1594287814679644276013]
? bnr2 = bnrinit(bnf,p,,2); \\ now fast
time = 1 ms.
? bnrclassfield(bnr2, 2)
%5 = [x^2 - 1594287814679644276013]
```

This will save a lot of time when the modulus contains a maximal ideal whose residue field is large.

bnrclassno(A, B, C)

Let A, B, C define a class field L over a ground field K (of type [:emphasis:`bnr]`, [:emphasis:`bnr, subgroup]`, or [:emphasis:`bnf, modulus]`, or [:emphasis:`bnf, modulus,:emphasis:subgroup]`, CFT (in the PARI manual)); this function returns the relative degree [L:K].

In particular if A is a *bnf* (with units), and B a modulus, this function returns the corresponding ray class number modulo B. One can input the attached *bid* (with generators if the subgroup C is non trivial) for B instead of the module itself, saving some time.

This function is faster than bnrinit and should be used if only the ray class number is desired. See bnrclassnolist if you need ray class numbers for all moduli less than some bound.

bnrclassnolist(bnf, list)

bnf being as output by bnfinit, and *list* being a list of moduli (with units) as output by ideallist or

ideallistarch, outputs the list of the class numbers of the corresponding ray class groups. To compute a single class number, bnrclassno is more efficient.

```
? bnf = bnfinit(x^2 - 2);
? L = ideallist(bnf, 100, 2);
? H = bnrclassnolist(bnf, L);
? H[98]
%4 = [1, 3, 1]
? l = L[1][98]; ids = vector(#l, i, l[i].mod[1])
%5 = [[98, 88; 0, 1], [14, 0; 0, 7], [98, 10; 0, 1]]
```

The weird l[i].mod[1], is the first component of l[i].mod, i.e. the finite part of the conductor. (This is cosmetic: since by construction the Archimedean part is trivial, I do not want to see it). This tells us that the ray class groups modulo the ideals of norm 98 (printed as %5) have respectively order 1, 3 and 1. Indeed, we may check directly:

? bnrclassno(bnf, ids[2])
%6 = 3

bnrconductor(A, B, C, flag)

Conductor f of the subfield of a ray class field as defined by [A, B, C] (of type [:emphasis:`bnr]`, [:emphasis:`bnr, *subgroup*]`, [:emphasis:`bnf, *modulus*]` or [:emphasis:`bnf, *modulus*, *subgroup*]`, CFT (in the PARI manual))

If flag = 0, returns f.

If flag = 1, returns $[f, Cl_f, H]$, where Cl_f is the ray class group modulo f, as a finite abelian group; finally H is the subgroup of Cl_f defining the extension.

If flag = 2, returns [f, bnr(f), H], as above except Cl_f is replaced by a bnr structure, as output by bnrinit(, f), without generators unless the input contained a *bnr* with generators.

In place of a subgroup H, this function also accepts a character $chi = (a_j)$, expressed as usual in terms of the generators bnr.gen: $\chi(g_j) = \exp(2i\pi a_j/d_j)$, where g_j has order $d_j = bnr.cyc[j]$. In which case, the function returns respectively

If flag = 0, the conductor f of $Ker\chi$.

If flag = 1, $[f, Cl_f, \chi_f]$, where χ_f is χ expressed on the minimal ray class group, whose modulus is the conductor.

If flag = 2, $[f, bnr(f), \chi_f]$.

Note. Using this function with flag! = 0 is usually a bad idea and kept for compatibility and convenience only: flag = 1 has always been useless, since it is no faster than flag = 2 and returns less information; flag = 2 is mostly OK with two subtle drawbacks:

- it returns the full *bnr* attached to the full ray class group, whereas in applications we only need Cl_f modulo N-th powers, where N is any multiple of the exponent of Cl_f/H . Computing directly the conductor, then calling bnrinit with optional argument N avoids this problem.
- computing the *bnr* needs only be done once for each conductor, which is not possible using this function.

For maximal efficiency, the recommended procedure is as follows. Starting from data (character or congruence subgroups) attached to a modulus m, we can first compute the conductors using this function with default flag = 0. Then for all data with a common conductor f || m, compute (once!) the *bnr* attached to f using bnrinit (modulo N-th powers for a suitable N!) and finally map original data to the new *bnr* using bnrmap.

bnrconductorofchar(bnr, chi)

This function is obsolete, use *bnrconductor*.

bnrdisc(A, B, C, flag)

A, B, C defining a class field L over a ground field K (of type [:emphasis:`bnr]`, [:emphasis:`bnr, subgroup]`, [:emphasis:`bnr, character]`, [:emphasis:`bnf, modulus]` or [:emphasis:`bnf, modulus, subgroup]`, CFT (in the PARI manual)), outputs data $[N, r_1, D]$ giving the discriminant and signature of L, depending on the binary digits of flag:

- 1: if this bit is unset, output absolute data related to L/Q: N is the absolute degree [L : Q], r₁ the number of real places of L, and D the discriminant of L/Q. Otherwise, output relative data for L/K: N is the relative degree [L : K], r₁ is the number of real places of K unramified in L (so that the number of real places of L is equal to r₁ times N), and D is the relative discriminant ideal of L/K.
- 2: if this bit is set and if the modulus is not the conductor of L, only return 0.

bnrdisclist(bnf, bound, arch)

bnf being as output by bnfinit (with units), computes a list of discriminants of Abelian extensions of the number field by increasing modulus norm up to bound *bound*. The ramified Archimedean places are given by *arch*; all possible values are taken if *arch* is omitted.

The alternative syntax bnrdisclist(bnf, list) is supported, where *list* is as output by ideallist or ideallistarch (with units), in which case *arch* is disregarded.

The output v is a vector, where v[k] is itself a vector w, whose length is the number of ideals of norm k.

- We consider first the case where *arch* was specified. Each component of w corresponds to an ideal m of norm k, and gives invariants attached to the ray class field L of bnf of conductor [m, arch]. Namely, each contains a vector [m, d, r, D] with the following meaning: m is the prime ideal factorization of the modulus, d = [L : ℚ] is the absolute degree of L, r is the number of real places of L, and D is the factorization of its absolute discriminant. We set d = r = D = 0 if m is not the finite part of a conductor.
- If arch was omitted, all $t = 2^{r_1}$ possible values are taken and a component of w has the form $[m, [[d_1, r_1, D_1], ..., [d_t, r_t, D_t]]]$, where m is the finite part of the conductor as above, and $[d_i, r_i, D_i]$ are the invariants of the ray class field of conductor $[m, v_i]$, where v_i is the *i*-th Archimedean component, ordered by inverse lexicographic order; so $v_1 = [0, ..., 0]$, $v_2 = [1, 0..., 0]$, etc. Again, we set $d_i = r_i = D_i = 0$ if $[m, v_i]$ is not a conductor.

Finally, each prime ideal $pr = [p, \alpha, e, f, \beta]$ in the prime factorization m is coded as the integer $p.n^2 + (f - 1).n + (j - 1)$, where n is the degree of the base field and j is such that

pr = idealprimedec(:emphasis:`nf,p)[j]`.

m can be decoded using **bnfdecodemodule**.

Note that to compute such data for a single field, either bnrclassno or bnrdisc are (much) more efficient.

bnrgaloisapply(bnr, mat, H)

Apply the automorphism given by its matrix *mat* to the congruence subgroup H given as a HNF matrix. The matrix *mat* can be computed with bnrgaloismatrix.

bnrgaloismatrix(bnr, aut)

Return the matrix of the action of the automorphism *aut* of the base field bnf.nf on the generators of the ray class field bnr.gen; *aut* can be given as a polynomial, an algebraic number, or a vector of automorphisms or a Galois group as output by galoisinit, in which case a vector of matrices is returned (in the later case, only for the generators aut.gen).

The generators bnr.gen need not be explicitly computed in the input *bnr*, which saves time: the result is well defined in this case also.

```
? K = bnfinit(a^4-3*a^2+253009); B = bnrinit(K,9); B.cyc
%1 = [8400, 12, 6, 3]
```

```
? G = nfgaloisconj(K)
%2 = [-a, a, -1/503*a^3 + 3/503*a, 1/503*a^3 - 3/503*a]~
? bnrgaloismatrix(B, G[2]) \\ G[2] = Id ...
%3 =
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
? bnrgaloismatrix(B, G[3]) \\ automorphism of order 2
%4 =
[799 0 0 2800]
[0704]
[4052]
[0002]
? M = %^2; for (i=1, #B.cyc, M[i,] %= B.cyc[i]); M
\%5 = \ acts on ray class group as automorphism of order 2
[1 \ 0 \ 0 \ 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

See bnrisgalois for further examples.

bnrinit(bnf, f, flag, cycmod)

bnf is as output by bnfinit (including fundamental units), f is a modulus, initializes data linked to the ray class group structure corresponding to this module, a so-called bnr structure. One can input the attached *bid* with generators for f instead of the module itself, saving some time. (As in idealstar, the finite part of the conductor may be given by a factorization into prime ideals, as produced by idealfactor.)

If the positive integer cycmod is present, only compute the ray class group modulo cycmod, which may save a lot of time when some maximal ideals in f have a huge residue field. In applications, we are given a congruence subgroup H and study the class field attached to Cl_f/H . If that finite Abelian group has an exponent which divides cycmod, then we have changed nothing theoretically, while trivializing expensive discrete logs in residue fields (since computations can be made modulo cycmod-th powers). This is useful in bnrclassfield, for instance when computing p-elementary extensions.

The following member functions are available on the result: .bnf is the underlying *bnf*, .mod the modulus, .bid the bid structure attached to the modulus; finally, .clgp, .no, .cyc, .gen refer to the ray class group (as a finite abelian group), its cardinality, its elementary divisors, its generators (only computed if flag = 1).

The last group of functions are different from the members of the underlying *bnf*, which refer to the class group; use :emphasis:`bnr.bnf.:emphasis:xxx` to access these, e.g. :emphasis:`bnr.bnf.cyc` to get the cyclic decomposition of the class group.

They are also different from the members of the underlying *bid*, which refer to $(\mathbb{Z}_K/f)^*$; use :emphasis:`bnr.bid.:emphasis:xxx`to access these, e.g. :emphasis:`bnr.bid.no`to get $\phi(f)$.

If flag = 0 (default), the generators of the ray class group are not explicitly computed, which saves time. Hence :emphasis:`bnr.gen` would produce an error. Note that implicit generators are still fixed and stored in the *bnr* (and guaranteed to be the same for fixed *bnf* and *bid* inputs), in terms of bnr.bnf.gen and bnr.bid.gen. The computation which is not performed is the expansion of such products in the ray class group so as to fix eplicit ideal representatives.

If flag = 1, as the default, except that generators are computed.

bnrisconductor(A, B, C)

Fast variant of bnrconductor(A, B, C); A, B, C represent an extension of the base field, given by class field theory (see CFT (in the PARI manual)). Outputs 1 if this modulus is the conductor, and 0 otherwise. This is slightly faster than bnrconductor when the character or subgroup is not primitive.

bnrisgalois(bnr, gal, H)

Check whether the class field attached to the subgroup H is Galois over the subfield of bnr.nf fixed by the group gal, which can be given as output by galoisinit, or as a matrix or a vector of matrices as output by bnrgaloismatrix, the second option being preferable, since it saves the recomputation of the matrices. Note: The function assumes that the ray class field attached to bnr is Galois, which is not checked.

In the following example, we lists the congruence subgroups of subextension of degree at most 3 of the ray class field of conductor 9 which are Galois over the rationals.

```
? K = bnfinit(a^4-3*a^2+253009); B = bnrinit(K,9); G = galoisinit(K);
? [H | H<-subgrouplist(B,3), bnrisgalois(B,G,H)];
time = 160 ms.
? M = bnrgaloismatrix(B,G);
? [H | H<-subgrouplist(B,3), bnrisgalois(B,M,H)]
time = 1 ms.
```

The second computation is much faster since bnrgaloismatrix(B,G) is computed only once.

bnrisprincipal(bnr, x, flag)

Let *bnr* be the ray class group data output by $\mathtt{bnrinit}(,, 1)$ and let x be an ideal in any form, coprime to the modulus f = bnr.mod. Solves the discrete logarithm problem in the ray class group, with respect to the generators bnr.gen, in a way similar to bnfisprincipal. If x is not coprime to the modulus of *bnr* the result is undefined. Note that *bnr* need not contain the ray class group generators, i.e. it may be created with $\mathtt{bnrinit}(,,0)$; in that case, although $\mathtt{bnr.gen}$ is undefined, we can still fix natural generators for the ray class group (in terms of the generators in $\mathtt{bnr.bnf.gen}$ and $\mathtt{bnr.bid.gen}$) and compute with respect to them.

The binary digits of flag (default flag = 1) mean:

- 1: If set returns a 2-component vector [e, α] where e is the vector of components of x on the ray class group generators, α is an element congruent to 1 mod*f such that x = α Π_i g_i^{e_i}. If unset, returns only e.
- 4: If set, returns $[e, \alpha]$ where α is given in factored form (compact representation). This is orders of magnitude faster.

```
? K = bnfinit(x^2 - 30); bnr = bnrinit(K, [4, [1,1]]);
? bnr.clgp \\ ray class group is isomorphic to Z/4 x Z/2 x Z/2
%2 = [16, [4, 2, 2]]
? P = idealprimedec(K, 3)[1]; \\ the ramified prime ideal above 3
? bnrisprincipal(bnr,P) \\ bnr.gen undefined !
%5 = [[3, 0, 0]~, 9]
? bnrisprincipal(bnr,P, 0) \\ omit principal part
%5 = [3, 0, 0]~
? bnr = bnrinit(bnr, bnr.bid, 1); \\ include explicit generators
```

```
? bnrisprincipal(bnr,P) \\ ... alpha is different !
%7 = [[3, 0, 0]~, 1/128625]
```

It may be surprising that the generator α is different although the underlying *bnf* and *bid* are the same. This defines unique generators for the ray class group as ideal *classes*, whether we use bnrinit(,0) or bnrinit(,1). But the actual ideal representatives (implicit if the flag is 0, computed and stored in the *bnr* if the flag is 1) are in general different and this is what happens here. Indeed, the implicit generators are naturally expressed expressed in terms of bnr.bnf.gen and bnr.bid.gen and *then* expanded and simplified (in the same ideal class) so that we obtain ideal representatives for bnr.gen which are as simple as possible. And indeed the quotient of the two α found is 1 modulo the conductor (and positive at the infinite places it contains), and this is the only guaranteed property.

Beware that, when bnr is generated using bnrinit(, cycmod), the results are given in Cl_f modulo cycmod-th powers:

```
? bnr2 = bnrinit(K, bnr.mod,, 2); \\ modulo squares
? bnr2.clgp
%9 = [8, [2, 2, 2]] \\ bnr.clgp tensored by Z/2Z
? bnrisprincipal(bnr2,P, 0)
%10 = [1, 0, 0]~
```

bnrmap(A, B)

This function has two different uses:

• if A and B are *bnr* structures for the same *bnf* attached to moduli m_A and m_B with $m_B || m_A$, return the canonical surjection from A to B, i.e. from the ray class group modulo m_A to the ray class group modulo m_B . The map is coded by a triple $[M, cyc_A, cyc_B]$: M gives the image of the fixed ray class group generators of A in terms of the ones in B, cyc_A and cyc_B are the cyclic structures A.cyc and B.cyc respectively. Note that this function does *not* need A or B to contain explicit generators for the ray class groups: they may be created using bnrinit(,0).

If B is only known modulo N-th powers (from bnrinit(,N)), the result is correct provided N is a multiple of the exponent of A.

• if A is a projection map as above and B is either a congruence subgroup H, or a ray class character χ , or a discrete logarithm (from bnrisprincipal) modulo m_A whose conductor divides m_B , return the image of the subgroup (resp. the character, the discrete logarighm) as defined modulo m_B . The main use of this variant is to compute the primitive subgroup or character attached to a *bnr* modulo their conductor. This is more efficient than bnrconductor in two respects: the *bnr* attached to the conductor need only be computed once and, most importantly, the ray class group can be computed modulo N-th powers, where N is a multiple of the exponent of Cl_{m_A}/H (resp. of the order of χ). Whereas bnrconductor is specified to return a *bnr* attached to the full ray class group, which may lead to untractable discrete logarithms in the full ray class group instead of a tiny quotient.

bnrrootnumber(bnr, chi, flag, precision)

If $\chi = chi$ is a character over *bnr*, not necessarily primitive, let $L(s, \chi) = \sum_{id} \chi(id)N(id)^{-s}$ be the attached Artin L-function. Returns the so-called Artin root number, i.e. the complex number $W(\chi)$ of modulus 1 such that

$$\Lambda(1-s,\chi) = W(\chi)\Lambda(s,\overline{\chi})$$

where $\Lambda(s,\chi) = A(\chi)^{s/2} \gamma_{\chi}(s) L(s,\chi)$ is the enlarged L-function attached to L.

You can set flag = 1 if the character is known to be primitive. Example:

```
bnf = bnfinit(x<sup>2</sup> - x - 57);
bnr = bnrinit(bnf, [7,[1,1]]);
bnrrootnumber(bnr, [2,1])
```

returns the root number of the character χ of $\operatorname{Cl}_{7oo_1oo_2}(\mathbb{Q}(\sqrt{229}))$ defined by $\chi(g_1^a g_2^b) = \zeta_1^{2a} \zeta_2^b$. Here g_1, g_2 are the generators of the ray-class group given by bnr.gen and $\zeta_1 = e^{2i\pi/N_1}, \zeta_2 = e^{2i\pi/N_2}$ where N_1, N_2 are the orders of g_1 and g_2 respectively ($N_1 = 6$ and $N_2 = 3$ as bnr.cyc readily tells us).

bnrstark(bnr, subgroup, precision)

bnr being as output by bnrinit, finds a relative equation for the class field corresponding to the modulus in *bnr* and the given congruence subgroup (as usual, omit *subgroup* if you want the whole ray class group).

The main variable of *bnr* must not be x, and the ground field and the class field must be totally real. When the base field is \mathbb{Q} , the vastly simpler galoissubcyclo is used instead. Here is an example:

bnf = bnfinit(y² - 3); bnr = bnrinit(bnf, 5); bnrstark(bnr)

returns the ray class field of $\mathbb{Q}(\sqrt{3})$ modulo 5. Usually, one wants to apply to the result one of

rnfpolredbest(bnf, pol) \\ compute a reduced relative polynomial
rnfpolredbest(bnf, pol, 2) \\ compute a reduced absolute polynomial

The routine uses Stark units and needs to find a suitable auxiliary conductor, which may not exist when the class field is not cyclic over the base. In this case bnrstark is allowed to return a vector of polynomials defining *independent* relative extensions, whose compositum is the requested class field. We decided that it was useful to keep the extra information thus made available, hence the user has to take the compositum herself, see nfcompositum.

Even if it exists, the auxiliary conductor may be so large that later computations become unfeasible. (And of course, Stark's conjecture may simply be wrong.) In case of difficulties, try bnrclassfield:

```
? bnr = bnrinit(bnfinit(y^8-12*y^6+36*y^4-36*y^2+9,1), 2);
? bnrstark(bnr)
*** at top-level: bnrstark(bnr)
*** ^_____
*** bnrstark: need 3919350809720744 coefficients in initzeta.
*** Computation impossible.
? bnrclassfield(bnr)
time = 20 ms.
%2 = [x^2 + (-2/3*y^6 + 7*y^4 - 14*y^2 + 3)]
```

call(*f*, *A*)

 $A = [a_1, ..., a_n]$ being a vector and f being a function, returns the evaluation of $f(a_1, ..., a_n)$. f can also be the name of a built-in GP function. If #A = 1, call (f, A) = apply (f, A)[1]. If f is variadic, the variadic arguments must grouped in a vector in the last component of A.

This function is useful

• when writing a variadic function, to call another one:

fprintf(file,format,args[..]) = write(file,call(strprintf,[format,args]))

• when dealing with function arguments with unspecified arity

The function below implements a global memoization interface:
```
memo=Map();
memoize(f,A[..])=
{
    my(res);
    if(!mapisdefined(memo, [f,A], &res),
    res = call(f,A);
    mapput(memo,[f,A],res));
    res;
}
```

for example:

```
? memoize(factor,2^128+1)
%3 = [59649589127497217,1;5704689200685129054721,1]
? ##
    *** last result computed in 76 ms.
? memoize(factor,2^128+1)
%4 = [59649589127497217,1;5704689200685129054721,1]
? ##
    *** last result computed in 0 ms.
? memoize(ffinit,3,3)
%5 = Mod(1,3)*x^3+Mod(1,3)*x^2+Mod(1,3)*x+Mod(2,3)
? fibo(n)=if(n==0,0,n==1,1,memoize(fibo,n-2)+memoize(fibo,n-1));
? fibo(100)
%7 = 354224848179261915075
```

• to call operators through their internal names without using alias

```
matnbelts(M) = call("_*_",matsize(M))
```

ceil(x)

Ceiling of x. When x is in \mathbb{R} , the result is the smallest integer greater than or equal to x. Applied to a rational function, ceil(x) returns the Euclidean quotient of the numerator by the denominator.

centerlift(x, v)

Same as lift, except that t_INTMOD and t_PADIC components are lifted using centered residues:

- for a t_INTMOD $x \in \mathbb{Z}/n\mathbb{Z}$, the lift y is such that -n/2 < y <= n/2.
- a t_PADIC x is lifted in the same way as above (modulo $p^padicprec(x)$) if its valuation v is nonnegative; if not, returns the fraction p^v centerlift (xp^{-v}) ; in particular, rational reconstruction is not attempted. Use bestappr for this.

For backward compatibility, centerlift(x, 'v) is allowed as an alias for lift(x, 'v).

characteristic(x)

Returns the characteristic of the base ring over which x is defined (as defined by t_INTMOD and t_FFELT components). The function raises an exception if incompatible primes arise from t_FFELT and t_PADIC components.

```
? characteristic(Mod(1,24)*x + Mod(1,18)*y)
%1 = 6
```

charconj(cyc, chi)

Let cyc represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{i \le k} \mathbb{Z}/d_j\mathbb{Z}$ with

 $d_k|...||d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the conjugate character.

? cyc = [15,5]; chi = [1,1]; ? charconj(cyc, chi) %2 = [14, 4] ? bnf = bnfinit(x^2+23); ? bnf.cyc %4 = [3] ? charconj(bnf, [1]) %5 = [2]

For Dirichlet characters (when cyc is znstar(q,1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character:

```
? G = znstar(8, 1); \\ (Z/8Z)^*
? charorder(G, 3) \\ Conrey label
%2 = 2
? chi = znconreylog(G, 3);
? charorder(G, chi) \\ Conrey logarithm
%4 = 2
```

chardiv(cyc, a, b)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given two characters a and b, return the character $a/b = a\overline{b}$.

```
? cyc = [15,5]; a = [1,1]; b = [2,4];
? chardiv(cyc, a,b)
%2 = [14, 2]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? chardiv(bnf, [1], [2])
%5 = [2]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character. If the two characters are in the same format, the result is given in the same format, otherwise a Conrey logarithm is used.

```
? G = znstar(100, 1);
? G.cyc
%2 = [20, 2]
? a = [10, 1]; \\ usual representation for characters
? b = 7; \\ Conrey label;
? c = znconreylog(G, 11); \\ Conrey log
? chardiv(G, b,b)
%6 = 1 \\ Conrey label
? chardiv(G, a,b)
```

%7 = [0, 5]~ \\ Conrey log
? chardiv(G, a,c)
%7 = [0, 14]~ \\ Conrey log

chareval(G, chi, x, z)

Let G be an abelian group structure affording a discrete logarithm method, e.g G = znstar(N, 1) for $(\mathbb{Z}/N\mathbb{Z})^*$ or a bnr structure, let x be an element of G and let chi be a character of G (see the note below for details). This function returns the value of chi at x.

Note on characters. Let K be some field. If G is an abelian group, let $\chi : G \to K^*$ be a character of finite order and let o be a multiple of the character order such that $\chi(n) = \zeta^{c(n)}$ for some fixed $\zeta \in K^*$ of multiplicative order o and a unique morphism $c : G \to (\mathbb{Z}/o\mathbb{Z}, +)$. Our usual convention is to write

$$G = (\mathbb{Z}/o_1\mathbb{Z})g_1 \oplus \ldots \oplus (\mathbb{Z}/o_d\mathbb{Z})g_d$$

for some generators (g_i) of respective order d_i , where the group has exponent $o := lcm_i o_i$. Since $\zeta^o = 1$, the vector (c_i) in $\prod (\mathbb{Z}/o_i\mathbb{Z})$ defines a character χ on G via $\chi(g_i) = \zeta^{c_i(o/o_i)}$ for all i. Classical Dirichlet characters have values in $K = \mathbb{C}$ and we can take $\zeta = \exp(2i\pi/o)$.

Note on Dirichlet characters. In the special case where *bid* is attached to $G = (\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), the Dirichlet character *chi* can be written in one of the usual 3 formats: a t_VEC in terms of bid.gen as above, a t_COL in terms of the Conrey generators, or a t_INT (Conrey label); see dirichletchar (in the PARI manual) or ??character.

The character value is encoded as follows, depending on the optional argument z:

- If z is omitted: return the rational number c(x)/o for x coprime to q, where we normalize $0 \le c(x) \le o$. If x can not be mapped to the group (e.g. x is not coprime to the conductor of a Dirichlet or Hecke character) we return the sentinel value -1.
- If z is an integer o, then we assume that o is a multiple of the character order and we return the integer c(x) when x belongs to the group, and the sentinel value -1 otherwise.
- z can be of the form [zeta, o], where zeta is an o-th root of 1 and o is a multiple of the character order. We return $\zeta^{c(x)}$ if x belongs to the group, and the sentinel value 0 otherwise. (Note that this coincides with the usual extension of Dirichlet characters to \mathbb{Z} , or of Hecke characters to general ideals.)
- Finally, z can be of the form [vzeta, o], where vzeta is a vector of powers $\zeta^0, ..., \zeta^{o-1}$ of some o-th root of 1 and o is a multiple of the character order. As above, we return $\zeta^{c(x)}$ after a table lookup. Or the sentinel value 0.

chargalois(cyc, ORD)

Let cyc represent a finite abelian group by its elementary divisors (any object which has a .cyc method is also allowed, i.e. the output of znstar or bnrinit). Return a list of representatives for the Galois orbits of complex characters of G. If ORD is present, select characters depending on their orders:

- if ORD is a t_INT, restrict to orders less than this bound;
- if ORD is a t_VEC or t_VECSMALL, restrict to orders in the list.

```
? G = znstar(96);
? #chargalois(G) \\ 16 orbits of characters mod 96
%2 = 16
? #chargalois(G,4) \\ order less than 4
%3 = 12
? chargalois(G,[1,4]) \\ order 1 or 4; 5 orbits
%4 = [[0, 0, 0], [2, 0, 0], [2, 1, 0], [2, 0, 1], [2, 1, 1]]
```

Given a character χ , of order n (charorder(G, chi)), the elements in its orbit are the $\phi(n)$ characters χ^i , (i, n) = 1.

charker(cyc, chi)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j \le k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the kernel of χ , as a matrix K in HNF which is a left-divisor of matdiagonal(d). Its columns express in terms of the g_i the generators of the subgroup. The determinant of K is the kernel index.

```
? cyc = [15,5]; chi = [1,1];
? charker(cyc, chi)
%2 =
[15 12]
[ 0 1]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charker(bnf, [1])
%5 =
[3]
```

Note that for Dirichlet characters (when cyc is znstar(q, 1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character.

? G = znstar(8, 1); \\ (Z/8Z)^**
? charker(G, 1) \\ Conrey label for trivial character
%2 =
[1 0]
[0 1]

charmul(cyc, a, b)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given two characters a and b, return the product character ab.

```
? cyc = [15,5]; a = [1,1]; b = [2,4];
? charmul(cyc, a,b)
%2 = [3, 0]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charmul(bnf, [1], [2])
%5 = [0]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character. If the two characters are in the same format, their product is given in the same format, otherwise a Conrey logarithm is used.

```
? G = znstar(100, 1);
? G.cyc
%2 = [20, 2]
? a = [10, 1]; \\ usual representation for characters
? b = 7; \\ Conrey label;
? c = znconreylog(G, 11); \\ Conrey log
? charmul(G, b,b)
%6 = 49 \\ Conrey label
? charmul(G, a,b)
%7 = [0, 15]~ \\ Conrey log
? charmul(G, a,c)
%7 = [0, 6]~ \\ Conrey log
```

charorder(cyc, chi)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j \le k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the order of the character chi.

```
? cyc = [15,5]; chi = [1,1];
? charorder(cyc, chi)
%2 = 15
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charorder(bnf, [1])
%5 = 3
```

For Dirichlet characters (when cyc is znstar(q, 1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character:

```
? G = znstar(100, 1); \\ (Z/100Z)^*
? charorder(G, 7) \\ Conrey label
%2 = 4
```

charpoly(A, v, flag)

characteristic polynomial of A with respect to the variable v, i.e. determinant of v * I - A if A is a square matrix.

? charpoly([1,2;3,4]); %1 = x^2 - 5*x - 2 ? charpoly([1,2;3,4],, 't) %2 = t^2 - 5*t - 2

If A is not a square matrix, the function returns the characteristic polynomial of the map "multiplication by A" if A is a scalar:

```
? charpoly(Mod(x+2, x^3-2))
%1 = x^3 - 6*x^2 + 12*x - 10
? charpoly(I)
%2 = x^2 + 1
? charpoly(quadgen(5))
```

%3 = x^2 - x - 1
? charpoly(ffgen(ffinit(2,4)))
%4 = Mod(1, 2)*x^4 + Mod(1, 2)*x^3 + Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2)

The value of flag is only significant for matrices, and we advise to stick to the default value. Let n be the dimension of A.

If flag = 0, same method (Le Verrier's) as for computing the adjoint matrix, i.e. using the traces of the powers of A. Assumes that n! is invertible; uses $O(n^4)$ scalar operations.

If flag = 1, uses Lagrange interpolation which is usually the slowest method. Assumes that n! is invertible; uses $O(n^4)$ scalar operations.

If flag = 2, uses the Hessenberg form. Assumes that the base ring is a field. Uses $O(n^3)$ scalar operations, but suffers from coefficient explosion unless the base field is finite or \mathbb{R} .

If flag = 3, uses Berkowitz's division free algorithm, valid over any ring (commutative, with unit). Uses $O(n^4)$ scalar operations.

If flag = 4, x must be integral. Uses a modular algorithm: Hessenberg form for various small primes, then Chinese remainders.

If flag = 5 (default), uses the "best" method given x. This means we use Berkowitz unless the base ring is \mathbb{Z} (use flag = 4) or a field where coefficient explosion does not occur, e.g. a finite field or the reals (use flag = 2).

charpow(*cyc*, *a*, *n*)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given $n \in \mathbb{Z}$ and a character a, return the character a^n .

```
? cyc = [15,5]; a = [1,1];
? charpow(cyc, a, 3)
%2 = [3, 3]
? charpow(cyc, a, 5)
%2 = [5, 0]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charpow(bnf, [1], 3)
%5 = [0]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character and the output uses the same format as the input.

```
? G = znstar(100, 1);
? G.cyc
%2 = [20, 2]
? a = [10, 1]; \\ standard representation for characters
? b = 7; \\ Conrey label;
? c = znconreylog(G, 11); \\ Conrey log
? charpow(G, a,3)
%6 = [10, 1] \\ standard representation
? charpow(G, b,3)
```

%7 = 43 \\ Conrey label
? charpow(G, c,3)
%8 = [1, 8]~ \\ Conrey log

chinese(x, y)

If x and y are both intmods or both polmods, creates (with the same type) a z in the same residue class as x and in the same residue class as y, if it is possible.

? chinese(Mod(1,2), Mod(2,3))
%1 = Mod(5, 6)
? chinese(Mod(x,x^2-1), Mod(x+1,x^2+1))
%2 = Mod(-1/2*x^2 + x + 1/2, x^4 - 1)

This function also allows vector and matrix arguments, in which case the operation is recursively applied to each component of the vector or matrix.

```
? chinese([Mod(1,2),Mod(1,3)], [Mod(1,5),Mod(2,7)])
%3 = [Mod(1, 10), Mod(16, 21)]
```

For polynomial arguments in the same variable, the function is applied to each coefficient; if the polynomials have different degrees, the high degree terms are copied verbatim in the result, as if the missing high degree terms in the polynomial of lowest degree had been Mod(0, 1). Since the latter behavior is usually *not* the desired one, we propose to convert the polynomials to vectors of the same length first:

```
? P = x+1; Q = x^2+2*x+1;
? chinese(P*Mod(1,2), Q*Mod(1,3))
%4 = Mod(1, 3)*x^2 + Mod(5, 6)*x + Mod(3, 6)
? chinese(Vec(P,3)*Mod(1,2), Vec(Q,3)*Mod(1,3))
%5 = [Mod(1, 6), Mod(5, 6), Mod(4, 6)]
? Pol(%)
%6 = Mod(1, 6)*x^2 + Mod(5, 6)*x + Mod(4, 6)
```

If y is omitted, and x is a vector, chinese is applied recursively to the components of x, yielding a residue belonging to the same class as all components of x.

Finally chinese(x, x) = x regardless of the type of x; this allows vector arguments to contain other data, so long as they are identical in both vectors.

$\operatorname{cmp}(x, y)$

Gives the result of a comparison between arbitrary objects x and y (as -1, 0 or 1). The underlying order relation is transitive, the function returns 0 if and only if x == y. It has no mathematical meaning but satisfies the following properties when comparing entries of the same type:

- two t_INT s compare as usual (i.e. cmp(x, y) < 0 if and only if x < y);
- two t_VECSMALL s of the same length compare lexicographically;
- two t_STR s compare lexicographically.

In case all components are equal up to the smallest length of the operands, the more complex is considered to be larger. More precisely, the longest is the largest; when lengths are equal, we have matrix > vector > scalar. For example:

```
? cmp(1, 2)
%1 = -1
```

```
? cmp(2, 1)
%2 = 1
? cmp(1, 1.0) \\ note that 1 == 1.0, but (1===1.0) is false.
%3 = -1
? cmp(x + Pi, [])
%4 = -1
```

This function is mostly useful to handle sorted lists or vectors of arbitrary objects. For instance, if v is a vector, the construction vecsort(v, cmp) is equivalent to Set(v).

component(x, n)

Extracts the n - th-component of x. This is to be understood as follows: every PARI type has one or two initial code words. The components are counted, starting at 1, after these code words. In particular if x is a vector, this is indeed the n - th-component of x, if x is a matrix, the n - th column, if x is a polynomial, the n - th coefficient (i.e. of degree n - 1), and for power series, the n - th significant coefficient.

For polynomials and power series, one should rather use polcoeff, and for vectors and matrices, the [] operator. Namely, if x is a vector, then x[n] represents the n - th component of x. If x is a matrix, x[m,n] represents the coefficient of row m and column n of the matrix, x[m,] represents the m - th row of x, and x[,n] represents the n - th column of x.

Using of this function requires detailed knowledge of the structure of the different PARI types, and thus it should almost never be used directly. Some useful exceptions:

```
? x = 3 + 0(3^5);
? component(x, 2)
%2 = 81 \\ p^(p-adic accuracy)
? component(x, 1)
%3 = 3 \\ p
? q = Qfb(1,2,3);
? component(q, 1)
%5 = 1
```

concat(x, y)

Concatenation of x and y. If x or y is not a vector or matrix, it is considered as a one-dimensional vector. All types are allowed for x and y, but the sizes must be compatible. Note that matrices are concatenated horizontally, i.e. the number of rows stays the same. Using transpositions, one can concatenate them vertically, but it is often simpler to use matconcat.

```
? x = matid(2); y = 2*matid(2);
? concat(x,y)
%2 =
[1 0 2 0]
[0 1 0 2]
? concat(x~,y~)~
%3 =
[1 0]
[0 1]
[2 0]
```

[0 2] ? matconcat([x;y]) %4 = [1 0] [0 1] [2 0] [0 2]

To concatenate vectors sideways (i.e. to obtain a two-row or two-column matrix), use Mat instead, or matconcat:

```
? x = [1,2];
? y = [3,4];
? concat(x,y)
%3 = [1, 2, 3, 4]
? Mat([x,y]~)
%4 =
[1 2]
[3 4]
? matconcat([x;y])
%5 =
[1 2]
[3 4]
```

Concatenating a row vector to a matrix having the same number of columns will add the row to the matrix (top row if the vector is x, i.e. comes first, and bottom row otherwise).

The empty matrix [;] is considered to have a number of rows compatible with any operation, in particular concatenation. (Note that this is *not* the case for empty vectors [] or $[]\sim$.)

If y is omitted, x has to be a row vector or a list, in which case its elements are concatenated, from left to right, using the above rules.

```
? concat([1,2], [3,4])
%1 = [1, 2, 3, 4]
? a = [[1,2]~, [3,4]~]; concat(a)
%2 =
[1 3]
[2 4]
? concat([1,2; 3,4], [5,6]~)
%3 =
[1 2 5]
[3 4 6]
? concat([%, [7,8]~, [1,2,3,4]])
%5 =
```

(continued from previous page)

[1	2	5	7]									
[3	4	6	8]									
[1	2	3	4]									

conj(x)

Conjugate of x. The meaning of this is clear, except that for real quadratic numbers, it means conjugation in the real quadratic field. This function has no effect on integers, reals, intmods, fractions or p-adics. The only forbidden type is polmod (see **conjvec** for this).

conjvec(z, precision)

Conjugate vector representation of z. If z is a polmod, equal to Mod(a, T), this gives a vector of length degree(T) containing:

- the complex embeddings of z if T has rational coefficients, i.e. the a(r[i]) where r = polroots(T);
- the conjugates of z if T has some intmod coefficients;
- if z is a finite field element, the result is the vector of conjugates $[z, z^p, z^{p^2}, ..., z^{p^{n-1}}]$ where n = degree(T).

If z is an integer or a rational number, the result is z. If z is a (row or column) vector, the result is a matrix whose columns are the conjugate vectors of the individual elements of z.

content(x, D)

Computes the gcd of all the coefficients of x, when this gcd makes sense. This is the natural definition if x is a polynomial (and by extension a power series) or a vector/matrix. This is in general a weaker notion than the *ideal* generated by the coefficients:

? content(2*x+y)
%1 = 1 \\ = gcd(2,y) over Q[y]

If x is a scalar, this simply returns the absolute value of x if x is rational (t_INT or t_FRAC), and either 1 (inexact input) or x (exact input) otherwise; the result should be identical to gcd(x, 0).

The content of a rational function is the ratio of the contents of the numerator and the denominator. In recursive structures, if a matrix or vector *coefficient* x appears, the gcd is taken not with x, but with its content:

```
? content([ [2], 4*matid(3) ])
%1 = 2
```

The content of a t_VECSMALL is computed assuming the entries are signed integers.

The optional argument D allows to control over which ring we compute and get a more predictable behaviour:

- 1: we only consider the underlying Q-structure and the denominator is a (positive) rational number
- a simple variable, say 'x: all entries are considered as rational functions in K(x) for some field K and the content is an element of K.

```
? f = x + 1/y + 1/2;
? content(f) \\ as a t_POL in x
%2 = 1/(2*y)
? content(f, 1) \\ Q-content
%3 = 1/2
? content(f, y) \\ as a rational function in y
%4 = 1/2
```

? g = x^2*y + y^2*x; ? content(g, x) %6 = y ? content(g, y) %7 = x

contfrac(x, b, nmax)

Returns the row vector whose components are the partial quotients of the continued fraction expansion of x. In other words, a result $[a_0, ..., a_n]$ means that $x a_0 + 1/(a_1 + ... + 1/a_n)$. The output is normalized so that $a_n! = 1$ (unless we also have n = 0).

The number of partial quotients n + 1 is limited by nmax. If nmax is omitted, the expansion stops at the last significant partial quotient.

```
? \p19
realprecision = 19 significant digits
? contfrac(Pi)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2]
? contfrac(Pi,, 3) \\ n = 2
%2 = [3, 7, 15]
```

x can also be a rational function or a power series.

If a vector b is supplied, the numerators are equal to the coefficients of b, instead of all equal to 1 as above; more precisely, $x (1/b_0)(a_0 + b_1/(a_1 + ... + b_n/a_n))$; for a numerical continued fraction (x real), the a_i are integers, as large as possible; if x is a rational function, they are polynomials with deg $a_i = \text{deg } b_i + 1$. The length of the result is then equal to the length of b, unless the next partial quotient cannot be reliably computed, in which case the expansion stops. This happens when a partial remainder is equal to zero (or too small compared to the available significant digits for x a t_REAL).

A direct implementation of the numerical continued fraction contfrac(x,b) described above would be

```
\\ "greedy" generalized continued fraction
cf(x, b) =
{ my( a= vector(#b), t );

    x *= b[1];
    for (i = 1, #b,
    a[i] = floor(x);
    t = x - a[i]; if (!t || i == #b, break);
    x = b[i+1] / t;
    ); a;
}
```

There is some degree of freedom when choosing the a_i ; the program above can easily be modified to derive variants of the standard algorithm. In the same vein, although no builtin function implements the related Engel expansion (a special kind of Egyptian fraction decomposition: $x = 1/a_1 + 1/(a_1a_2) + ...$), it can be obtained as follows:

```
\\ n terms of the Engel expansion of x
engel(x, n = 10) =
{ my( u = x, a = vector(n) );
for (k = 1, n,
    a[k] = ceil(1/u);
```

```
u = u*a[k] - 1;
if (!u, break);
); a
}
```

Obsolete hack. (don't use this): if b is an integer, *nmax* is ignored and the command is understood as contfrac(:math:`x,, b)`.

contfraceval(CF, t, lim)

Given a continued fraction CF output by contfracinit, evaluate the first lim terms of the continued fraction at t (all terms if lim is negative or omitted; if positive, lim must be less than or equal to the length of CF.

contfracinit(M, lim)

Given M representing the power series $S = \sum_{n>=0} M[n+1]z^n$, transform it into a continued fraction in Euler form, using the quotient-difference algorithm; restrict to $n \le lim$ if latter is nonnegative. M can be a vector, a power series, a polynomial; if the limiting parameter lim is present, a rational function is also allowed (and converted to a power series of that accuracy).

The result is a 2-component vector [A, B] such that $S = M[1]/(1 + A[1]z + B[1]z^2/(1 + A[2]z + B[2]z^2/(1 + ...1/(1 + A[lim/2]z))))$. Does not work if any coefficient of M vanishes, nor for series for which certain partial denominators vanish.

contfracpnqn(x, n)

When x is a vector or a one-row matrix, x is considered as the list of partial quotients $[a_0, a_1, ..., a_n]$ of a rational number, and the result is the 2 by 2 matrix $[p_n, p_{n-1}; q_n, q_{n-1}]$ in the standard notation of continued fractions, so $p_n/q_n = a_0 + 1/(a_1 + ... + 1/a_n)$. If x is a matrix with two rows $[b_0, b_1, ..., b_n]$ and $[a_0, a_1, ..., a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = (1/b_0)(a_0+b_1/(a_1+...+b_n/a_n))$. Note that in this case one usually has $b_0 = 1$.

If $n \ge 0$ is present, returns all convergents from p_0/q_0 up to p_n/q_n . (All convergents if x is too small to compute the n + 1 requested convergents.)

```
? a = contfrac(Pi,10)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 3]
? allpnqn(x) = contfracpnqn(x,#x) \\ all convergents
? allpnqn(a)
%3 =
[3 22 333 355 103993 104348 208341 312689 1146408]
[1 7 106 113 33102 33215 66317 99532 364913]
? contfracpnqn(a) \\ last two convergents
%4 =
[1146408 312689]
[ 364913 99532]
? contfracpnqn(a,3) \\ first three convergents
%5 =
[3 22 333 355]
[1 7 106 113]
```

core(n, flag)

If n is an integer written as $n = df^2$ with d squarefree, returns d. If flag is nonzero, returns the two-element row vector [d, f]. By convention, we write $0 = 0x1^2$, so core(\emptyset , 1) returns [0, 1].

coredisc(n, flag)

A *fundamental discriminant* is an integer of the form $t = 1 \mod 4$ or $4t = 8, 12 \mod 16$, with t squarefree (i.e. 1 or the discriminant of a quadratic number field). Given a nonzero integer n, this routine returns the (unique) fundamental discriminant d such that $n = df^2$, f a positive rational number. If *flag* is nonzero, returns the two-element row vector [d, f]. If n is congruent to 0 or 1 modulo 4, f is an integer, and a half-integer otherwise.

By convention, coredisc(0, 1)) returns [0, 1].

Note that quaddisc(n) returns the same value as coredisc(n), and also works with rational inputs $n \in \mathbb{Q}^*$.

cos(*x*, *precision*)

Cosine of x. Note that, for real x, cosine and sine can be obtained simultaneously as

 $cs(x) = my(z = exp(I^*x)); [real(z), imag(z)];$

and for general complex x as

|--|

Note that the latter function suffers from catastrophic cancellation when z^2 1.

cosh(*x*, *precision*)

Hyperbolic cosine of x.

cotan(*x*, *precision*)

Cotangent of x.

cotanh(x, precision)

Hyperbolic cotangent of x.

default(key, val)

Returns the default corresponding to keyword *key*. If *val* is present, sets the default to *val* first (which is subject to string expansion first). Typing default() (or \d) yields the complete default list as well as their current values. See defaults (in the PARI manual) for an introduction to GP defaults, gp_defaults (in the PARI manual) for a list of available defaults, and meta (in the PARI manual) for some shortcut alternatives. Note that the shortcuts are meant for interactive use and usually display more information than default.

denominator(f, D)

Denominator of f. The meaning of this is clear when f is a rational number or function. If f is an integer or a polynomial, it is treated as a rational number or function, respectively, and the result is equal to 1. For polynomials, you probably want to use

```
denominator( content(f) )
```

instead. As for modular objects, t_INTMOD and t_PADIC have denominator 1, and the denominator of a t_POLMOD is the denominator of its lift.

If f is a recursive structure, for instance a vector or matrix, the lcm of the denominators of its components (a common denominator) is computed. This also applies for t_COMPLEX s and t_QUAD s.

Warning. Multivariate objects are created according to variable priorities, with possibly surprising side effects (x/y) is a polynomial, but y/x is a rational function). See priority (in the PARI manual).

The optional argument D allows to control over which ring we compute the denominator and get a more predictable behaviour:

- 1: we only consider the underlying Q-structure and the denominator is a (positive) rational integer
- a simple variable, say 'x: all entries as rational functions in K(x) and the denominator is a polynomial in x.

? f = x + 1/y + 1/2; ? denominator(f) \\ a t_POL in x %2 = 1 ? denominator(f, 1) \\ Q-denominator %3 = 2 ? denominator(f, x) \\ as a t_POL in x, seen above %4 = 1 ? denominator(f, y) \\ as a rational function in y %5 = 2*y

deriv(x, v)

Derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise. One can use x' as a shortcut if the derivative is with respect to the main variable of x; and also use x", etc., for multiple derivatives altough derivn is often preferrable.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the derivative is taken with respect to the main variable used in the base ring R.

```
? f = (x/y)^5;
? deriv(f)
%2 = 5/y^5*x^4
? f'
%3 = 5/y^5*x^4
? deriv(f, 'x) \\ same since 'x is the main variable
%4 = 5/y^5*x^4
? deriv(f, 'y)
%5 = -5/y^6*x^5
```

This function also operates on closures, in which case the variable must be omitted. It returns a closure performing a numerical differentiation as per derivnum:

derivn(*x*, *n*, *v*)

n-th derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise; the integer n must be nonnegative. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise. One can use x', x, etc., as a shortcut if the derivative is with respect to the main variable of x.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the derivative is taken with respect to the main variable used in the base ring R.

? f = (x/y)^5; ? derivn(f, 2) %2 = 20/y^5*x^3 ? f'' %3 = 20/y^5*x^3 ? derivn(f, 2, 'x) \\ same since 'x is the main variable %4 = 20/y^5*x^3 ? derivn(f, 2, 'y) %5 = 30/y^7*x^5

This function also operates on closures, in which case the variable must be omitted. It returns a closure performing a numerical differentiation as per derivnum:

diffop(x, v, d, n)

Let v be a vector of variables, and d a vector of the same length, return the image of x by the n-power (1 if n is not given) of the differential operator D that assumes the value d[i] on the variable v[i]. The value of D on a scalar type is zero, and D applies componentwise to a vector or matrix. When applied to a t_POLMOD, if no value is provided for the variable of the modulus, such value is derived using the implicit function theorem.

Examples. This function can be used to differentiate formal expressions: if $E = \exp(X^2)$ then we have E' = 2 * X * E. We derivate $X * exp(X^2)$ as follows:

? diffop(E*X,[X,E],[1,2*X*E]) %1 = (2*X^2 + 1)*E

Let Sin and Cos be two function such that $Sin^2 + Cos^2 = 1$ and Cos' = -Sin. We can differentiate Sin/Cos as follows, PARI inferring the value of Sin' from the equation:

? diffop(Mod('Sin/'Cos,'Sin^2+'Cos^2-1),['Cos],[-'Sin]) %1 = Mod(1/Cos^2, Sin^2 + (Cos^2 - 1))

Compute the Bell polynomials (both complete and partial) via the Faa di Bruno formula:

```
Bell(k,n=-1)=
{ my(x, v, dv, var = i->eval(Str("X",i)));
    v = vector(k, i, if (i==1, 'E, var(i-1)));
    dv = vector(k, i, if (i==1, 'X*var(1)*'E, var(i)));
    x = diffop('E,v,dv,k) / 'E;
    if (n < 0, subst(x,'X,1), polcoef(x,n,'X));
}</pre>
```

digits(x, b)

Outputs the vector of the digits of ||x|| in base b, where x and b are integers (b = 10 by default). For $x \ge 1$, the number of digits is logint(x, b) + 1. See from digits for the reverse operation.

? digits(1230) %1 = [1, 2, 3, 0] ? digits(10, 2) \\ base 2 %2 = [1, 0, 1, 0]

By convention, 0 has no digits:

```
? digits(0)
%3 = []
```

dilog(*x*, *precision*)

Principal branch of the dilogarithm of x, i.e. analytic continuation of the power series $\log_2(x) = \sum_{n>=1} x^n/n^2$.

dirdiv(x, y)

x and y being vectors of perhaps different lengths but with y[1]! = 0 considered as Dirichlet series, computes the quotient of x by y, again as a vector.

dirmul(x, y)

x and y being vectors of perhaps different lengths representing the Dirichlet series $\sum_n x_n n^{-s}$ and $\sum_n y_n n^{-s}$, computes the product of x by y, again as a vector.

? dirmul(vector(10,n,1), vector(10,n,moebius(n)))
%1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The product length is the minimum of # x * v(y) and # y * v(x), where v(x) is the index of the first nonzero coefficient.

```
? dirmul([0,1], [0,1]);
%2 = [0, 0, 0, 1]
```

dirpowers(*n*, *x*, *precision*)

For nonnegative n and complex number x, return the vector with n components $[1^x, 2^x, ..., n^x]$.

? dirpowers(5, 2) %1 = [1, 4, 9, 16, 25] ? dirpowers(5, 1/2) %2 = [1, 1.414..., 1.732..., 2.000..., 2.236...]

When $n \leq 0$, the function returns the empty vector [].

dirpowerssum(*n*, *x*, *precision*)

For positive integer n and complex number x, return the sum $1^x + 2^x + ... + n^x$. This is the same as vecsum(dirpowers(n,x)), but faster and using only $O(\sqrt{n})$ memory instead of O(n).

```
? dirpowers(5, 2)
%1 = [1, 4, 9, 16, 25]
? vecsum(%)
%2 = 55
? dirpowerssum(5, 2)
%3 = 55
? \p200
? dirpowerssum(10^7, 1/2 + I * sqrt(3));
time = 29,884 ms.
```

```
? vecsum(dirpowers(10^7, 1/2 + I * sqrt(3)))
time = 41,894 ms.
```

The penultimate command works with default stack size, the last one requires a stacksize of at least 5GB.

When $n \leq 0$, the function returns 0.

dirzetak(nf, b)

Gives as a vector the first b coefficients of the Dedekind zeta function of the number field nf considered as a Dirichlet series.

divisors(x, flag)

Creates a row vector whose components are the divisors of x. The factorization of x (as output by factor) can be used instead. If flag = 1, return pairs [d, factor(d)].

By definition, these divisors are the products of the irreducible factors of n, as produced by factor(n), raised to appropriate powers (no negative exponent may occur in the factorization). If n is an integer, they are the positive divisors, in increasing order.

```
? divisors(12)
%1 = [1, 2, 3, 4, 6, 12]
? divisors(12, 1) \\ include their factorization
%2 = [[1, matrix(0,2)], [2, Mat([2, 1])], [3, Mat([3, 1])],
[4, Mat([2, 2])], [6, [2, 1; 3, 1]], [12, [2, 2; 3, 1]]]
? divisors(x<sup>4</sup> + 2*x<sup>3</sup> + x<sup>2</sup>) \\ also works for polynomials
%3 = [1, x, x<sup>2</sup>, x + 1, x<sup>2</sup> + x, x<sup>3</sup> + x<sup>2</sup>, x<sup>2</sup> + 2*x + 1,
x<sup>3</sup> + 2*x<sup>2</sup> + x, x<sup>4</sup> + 2*x<sup>3</sup> + x<sup>2</sup>]
```

This function requires a lot of memory if x has many divisors. The following idiom runs through all divisors using very little memory, in no particular order this time:

```
F = factor(x); P = F[,1]; E = F[,2];
forvec(e = vectorv(#E,i,[0,E[i]]), d = factorback(P,e); ...)
```

If the factorization of d is also desired, then [P, e] almost provides it but not quite: e may contain 0 exponents, which are not allowed in factorizations. These must be sieved out as in:

```
tofact(P,E) =
  my(v = select(x->x, E, 1)); Mat([vecextract(P,v), vecextract(E,v)]);
? tofact([2,3,5,7]~, [4,0,2,0]~)
%4 =
[2 4]
[5 2]
```

We can then run the above loop with tofact(P,e) instead of, or together with, factorback.

divisorslenstra(N, r, s)

Given three integers N > s > r >= 0 such that (r, s) = 1 and $s^3 > N$, find all divisors d of N such that d = r(mods). There are at most 11 such divisors (Lenstra).

? N = 245784; r = 19; s = 65 ; ? divisorslenstra(N, r, s)

%2 = [19, 84, 539, 1254, 3724, 245784]
? [d | d <- divisors(N), d % s == r]
%3 = [19, 84, 539, 1254, 3724, 245784]</pre>

When the preconditions are not met, the result is undefined:

? N = 4484075232; r = 7; s = 1303; s^3 > N %4 = 0 ? divisorslenstra(N, r, s) ? [d | d <- divisors(N), d % s == r] %6 = [7, 2613, 9128, 19552, 264516, 3407352, 344928864]

(Divisors were missing but $s^3 < N$.)

divrem(x, y, v)

Creates a column vector with two components, the first being the Euclidean quotient (:math:`x \:math:y`), the second the Euclidean remainder (:math:`x - (x\:math:y)*:math:y`), of the division of x by y. This avoids the need to do two divisions if one needs both the quotient and the remainder. If v is present, and x, y are multivariate polynomials, divide with respect to the variable v.

Beware that divrem(:math:`x,:math:y)[2]` is in general not the same as :math:`x % y; no GP operator corresponds to it:

eint1(x, n, precision)

Exponential integral $\int_x^o o(e^{-t})/(t)dt = incgam(0, x)$, where the latter expression extends the function definition from real x > 0 to all complex x! = 0.

If n is present, we must have x > 0; the function returns the n-dimensional vector [eint1(x), ..., eint1(nx)]. Contrary to other transcendental functions, and to the default case (n omitted), the values are correct up to a bounded *absolute*, rather than relative, error 10^{-n} , where n is precision(x) if x is a t_REAL and defaults to realprecision otherwise. (In the most important application, to the computation of L-functions via approximate functional equations, those values appear as weights in long sums and small individual relative errors are less useful than controlling the absolute error.) This is faster than repeatedly calling eint1(:math:`i * x)`, but less precise.

ellE(k, precision)

Complete elliptic integral of the second kind

$$E(k) = \int_0^{\pi/2} (1 - k^2 \sin(t)^2)^{1/2} dt$$

for the complex parameter k using the agm.

ellK(k, precision)

Complete elliptic integral of the first kind

$$K(k) = \int_0^{\pi/2} (1 - k^2 \sin(t)^2)^{-1/2} dt$$

for the complex parameter k using the agm.

ellL1(*E*, *r*, *precision*)

Returns the value at s = 1 of the derivative of order r of the L-function of the elliptic curve E.

```
? E = ellinit("11a1"); \\ order of vanishing is 0
? ellL1(E)
%2 = 0.2538418608559106843377589233
? E = ellinit("389a1"); \\ order of vanishing is 2
? ellL1(E)
%4 = -5.384067311837218089235032414 E-29
? ellL1(E, 1)
%5 = 0
? ellL1(E, 2)
%6 = 1.518633000576853540460385214
```

The main use of this function, after computing at *low* accuracy the order of vanishing using ellanalyticrank, is to compute the leading term at *high* accuracy to check (or use) the Birch and Swinnerton-Dyer conjecture:

```
? \p18
realprecision = 18 significant digits
? E = ellinit("5077a1"); ellanalyticrank(E)
time = 8 ms.
%1 = [3, 10.3910994007158041]
? \p200
realprecision = 202 significant digits (200 digits displayed)
? ellL1(E, 3)
time = 104 ms.
%3 = 10.3910994007158041387518505103609170697263563756570092797[...]
```

elladd(*E*, *z*1, *z*2)

Sum of the points z1 and z2 on the elliptic curve corresponding to E.

ellak(E, n)

Computes the coefficient a_n of the *L*-function of the elliptic curve E/\mathbb{Q} , i.e. coefficients of a newform of weight 2 by the modularity theorem (Taniyama-Shimura-Weil conjecture). E must be an ell structure over \mathbb{Q} as output by ellipit. E must be given by an integral model, not necessarily minimal, although a minimal model will make the function faster.

```
? E = ellinit([1,-1,0,4,3]);
? ellak(E, 10)
%2 = -3
? e = ellchangecurve(E, [1/5,0,0,0]); \\ made not minimal at 5
? ellak(e, 10) \\ wasteful but works
%3 = -3
? E = ellminimalmodel(e); \\ now minimal
? ellak(E, 5)
%5 = -3
```

If the model is not minimal at a number of bad primes, then the function will be slower on those n divisible by the bad primes. The speed should be comparable for other n:

```
? for(i=1,10^6, ellak(E,5))
time = 699 ms.
? for(i=1,10^6, ellak(e,5)) \\ 5 is bad, markedly slower
time = 1,079 ms.
? for(i=1,10^5,ellak(E,5*i))
time = 1,477 ms.
? for(i=1,10^5,ellak(e,5*i)) \\ still slower but not so much on average
time = 1,569 ms.
```

ellan(E, n)

Computes the vector of the first *n* Fourier coefficients a_k corresponding to the elliptic curve *E* defined over a number field. If *E* is defined over \mathbb{Q} , the curve may be given by an arbitrary model, not necessarily minimal, although a minimal model will make the function faster. Over a more general number field, the model must be locally minimal at all primes above 2 and 3.

ellanalyticrank(E, eps, precision)

Returns the order of vanishing at s = 1 of the *L*-function of the elliptic curve *E* and the value of the first nonzero derivative. To determine this order, it is assumed that any value less than eps is zero. If eps is omitted, $2^{-b/2}$ is used, where *b* is the current bit precision.

```
? E = ellinit("11a1"); \\ rank 0
? ellanalyticrank(E)
%2 = [0, 0.2538418608559106843377589233]
? E = ellinit("37a1"); \\ rank 1
? ellanalyticrank(E)
%4 = [1, 0.3059997738340523018204836835]
? E = ellinit("389a1"); \\ rank 2
? ellanalyticrank(E)
%6 = [2, 1.518633000576853540460385214]
? E = ellinit("5077a1"); \\ rank 3
? ellanalyticrank(E)
%8 = [3, 10.39109940071580413875185035]
```

ellap(E, p)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. If the field $K = \mathbb{F}_q$ is finite, return the trace of Frobenius *t*, defined by the equation $\#E(\mathbb{F}_q) = q + 1 - t$.

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the trace of Frobenius for the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number $(K = \mathbb{Q})$ or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The equation need not be minimal or even integral at p; of course, a minimal model will be more efficient.

For a number field K, the trace of Frobenius is the a_p coefficient in the Euler product defining the curve L-series, whence the function name:

$$L(E/K,s) = \prod_{badp} (1 - a_p(Np)^{-s})^{-1} \prod_{goodp} (1 - a_p(Np)^{-s} + (Np)^{1-2s})^{-1}.$$

When the characteristic of the finite field is large, the availability of the seadata package will speed up the computation.

```
? E = ellinit([0,1]); \\ y^2 = x^3 + 0.x + 1, defined over Q
? ellap(E, 7) \setminus 7 necessary here
\%2 = -4 \setminus \#E(F_7) = 7+1-(-4) = 12
? ellcard(E, 7)
%3 = 12 \\ OK
? E = ellinit([0,1], 11); \setminus Aefined over F_{11}
? ellap(E) \setminus no need to repeat 11
\%4 = 0
? ellap(E, 11) \setminus ... but it also works
%5 = 0
? ellgroup(E, 13) \\ ouch, inconsistent input!
 *** at top-level: ellap(E,13)
*** ^_____
 *** ellap: inconsistent moduli in Rg_to_Fp:
 11
13
? a = ffgen(ffinit(11,3), 'a); \setminus defines F_q := F_{11^3}
? E = ellinit([a+1,a]); \ y^2 = x^3 + (a+1)x + a, defined over F_q
? ellap(E)
%8 = −3
```

If the curve is defined over a more general number field than \mathbb{Q} , the maximal ideal p must be explicitly given in idealprimedec format. There is no assumption of local minimality at p.

```
? K = nfinit(a^2+1); E = ellinit([1+a,0,1,0,0], K);
? fa = idealfactor(K, E.disc)
%2 =
[ [5, [-2, 1], 1, 1, [2, -1; 1, 2] ] 1 ]
[[13, [5, 1]~, 1, 1, [-5, -1; 1, -5]] 2]
? ellap(E, fa[1,1])
%3 = -1 \setminus  nonsplit multiplicative reduction
? ellap(E, fa[2,1])
%4 = 1 \setminus  split multiplicative reduction
? P17 = idealprimedec(K,17)[1];
? ellap(E, P17)
\%6 = 6 \setminus \text{good reduction}
? E2 = ellchangecurve(E, [17,0,0,0]);
? ellap(E2, P17)
\%8 = 6 \setminus \text{same, starting from a nonmiminal model}
? P3 = idealprimedec(K,3)[1];
? ellap(E, P3) \setminus OK: E is minimal at P3
\%10 = -2
? E3 = ellchangecurve(E, [3,0,0,0]);
? ellap(E3, P3) \\ not integral at P3
 *** at top-level: ellap(E3,P3)
 *** ellap: impossible inverse in Rg_to_ff: Mod(0, 3).
```

Algorithms used. If E/\mathbb{F}_q has CM by a principal imaginary quadratic order we use a fast explicit formula (involving essentially Kronecker symbols and Cornacchia's algorithm), in $O(\log q)^2$ bit operations. Otherwise, we use Shanks-Mestre's baby-step/giant-step method, which runs in time $O(q^{1/4})$ using $O(q^{1/4})$ storage, hence

becomes unreasonable when q has about 30 digits. Above this range, the SEA algorithm becomes available, heuristically in $O(\log q)^4$, and primes of the order of 200 digits become feasible. In small characteristic we use Mestre's (p = 2), Kohel's (p = 3,5,7,13), Satoh-Harley (all in $O(p^2n^2)$) or Kedlaya's (in $O(pn^3)$) algorithms.

ellbil(*E*, *z1*, *z2*, *precision*)

Deprecated alias for ellheight(E,P,Q).

ellbsd(E, precision)

The object E being an elliptic curve over a number field, returns a real number c such that the BSD conjecture predicts that $L_E^{(r)}(1)/r! = cRS$ where r is the rank, R the regulator and S the cardinal of the Tate-Shafarevich group.

```
? e = ellinit([0,-1,1,-10,-20]); \\ rank 0
? ellbsd(e)
%2 = 0.25384186085591068433775892335090946105
? lfun(e,1)
%3 = 0.25384186085591068433775892335090946104
? e = ellinit([0,0,1,-1,0]); \\ rank 1
? P = ellheegner(e);
? ellbsd(e)*ellheight(e,P)
%6 = 0.30599977383405230182048368332167647445
? lfun(e,1,1)
%7 = 0.30599977383405230182048368332167647445
? e = ellinit([1+a,0,1,0,0],nfinit(a^2+1)); \\ rank 0
? ellbsd(e)
%9 = 0.42521832235345764503001271536611593310
? lfun(e,1)
%10 = 0.42521832235345764503001271536611593309
```

ellcard(E, p)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. If $K = \mathbb{F}_q$ is finite, return the order of the group $E(\mathbb{F}_q)$.

? E = ellinit([-3,1], 5); ellcard(E) %1 = 7 ? t = ffgen(3^5,'t); E = ellinit([t,t^2+1]); ellcard(E) %2 = 217

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the order of the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number ($K = \mathbb{Q}$) or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The equation need not be minimal or even integral at p; of course, a minimal model will be more efficient. The function considers the group of nonsingular points of the reduction of a minimal model of the curve at p, so also makes sense when the curve has bad reduction.

```
? E = ellinit([-3,1]);
? factor(E.disc)
%2 =
[2 4]
[3 4]
? ellcard(E, 5) \\ as above !
%3 = 7
```

? ellcard(E, 2) \\ additive reduction
%4 = 2

When the characteristic of the finite field is large, the availability of the seadata package will speed the computation. See also ellap for the list of implemented algorithms.

ellchangecurve(E, v)

Changes the data for the elliptic curve E by changing the coordinates using the vector $\mathbf{v} = [\mathbf{u}, \mathbf{r}, \mathbf{s}, \mathbf{t}]$, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. E must be an ell structure as output by ellinit. The special case v = 1 is also used instead of [1, 0, 0, 0] to denote the trivial coordinate change.

ellchangepoint(x, v)

Changes the coordinates of the point or vector of points x using the vector $\mathbf{v} = [\mathbf{u}, \mathbf{r}, \mathbf{s}, \mathbf{t}]$, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (see also ellchangecurve).

```
? E0 = ellinit([1,1]); P0 = [0,1]; v = [1,2,3,4];
? E = ellchangecurve(E0, v);
? P = ellchangepoint(P0,v)
%3 = [-2, 3]
? ellisoncurve(E, P)
%4 = 1
? ellchangepointinv(P,v)
%5 = [0, 1]
```

ellchangepointinv(x, v)

Changes the coordinates of the point or vector of points x using the inverse of the isomorphism attached to v = [u,r,s,t], i.e. if x' and y' are the old coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (inverse of ellchangepoint).

? E0 = ellinit([1,1]); P0 = [0,1]; v = [1,2,3,4]; ? E = ellchangecurve(E0, v); ? P = ellchangepoint(P0,v) %3 = [-2, 3] ? ellisoncurve(E, P) %4 = 1 ? ellchangepointinv(P,v) %5 = [0, 1] \\ we get back P0

ellconvertname(name)

Converts an elliptic curve name, as found in the elldata database, from a string to a triplet [conductor, isogenyclass, index]. It will also convert a triplet back to a curve name. Examples:

? ellconvertname("123b1")
%1 = [123, 1, 1]
? ellconvertname(%)
%2 = "123b1"

elldivpol(*E*, *n*, *v*)

n-division polynomial f_n for the curve E in the variable v. In standard notation, for any affine point P = (X, Y) on the curve and any integer $n \ge 0$, we have

$$[n]P = (\phi_n(P)\psi_n(P):\omega_n(P):\psi_n(P)^3)$$

for some polynomials ϕ_n, ω_n, ψ_n in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][X, Y]$. We have $f_n(X) = \psi_n(X)$ for n odd, and $f_n(X) = \psi_n(X, Y)(2Y + a_1X + a_3)$ for n even. We have

$$f_0 = 0, f_1 = 1, f_2 = 4X^3 + b_2X^2 + 2b_4X + b_6, f_3 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8,$$

$$f_4 = f_2(2X^6 + b_2X^5 + 5b_4X^4 + 10b_6X^3 + 10b_8X^2 + (b_2b_8 - b_4b_6)X + (b_8b_4 - b_6^2)), \dots$$

When n is odd, the roots of f_n are the X-coordinates of the affine points in the n-torsion subgroup E[n]; when n is even, the roots of f_n are the X-coordinates of the affine points in E[n] E[2] when n > 2, resp. in E[2] when n = 2. For n < 0, we define $f_n := -f_{-n}$.

elleisnum(w, k, flag, precision)

k being an even positive integer, computes the numerical value of the Eisenstein series of weight k at the lattice w, as given by ellperiods, namely

$$(2i\pi/\omega_2)^k(1+2/\zeta(1-k)\sum_{n>=1}n^{k-1}q^n/(1-q^n)),$$

where $q = \exp(2i\pi\tau)$ and $\tau := \omega_1/\omega_2$ belongs to the complex upper half-plane. It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by ellinit.

When *flag* is nonzero and k = 4 or 6, returns the elliptic invariants g_2 or g_3 , such that

$$y^2 = 4x^3 - g_2x - g_3$$

is a Weierstrass equation for E.

elleta(w, precision)

Returns the quasi-periods $[\eta_1, \eta_2]$ attached to the lattice basis $w = [\omega_1, \omega_2]$. Alternatively, w can be an elliptic curve E as output by ellinit, in which case, the quasi periods attached to the period lattice basis :math: `E.omega` (namely, :math: `E.eta`) are returned.

```
? elleta([1, I])
%1 = [3.141592653589793238462643383, 9.424777960769379715387930149*I]
```

ellformaldifferential(E, serprec, n)

Let $\omega := dx/(2y + a_1x + a_3)$ be the invariant differential form attached to the model E of some elliptic curve (ellinit form), and $\eta := x(t)\omega$. Return n terms (seriesprecision by default) of f(t), g(t) two power series in the formal parameter t = -x/y such that $\omega = f(t)dt, \eta = g(t)dt$:

$$f(t) = 1 + a_1 t + (a_1^2 + a_2)t^2 + \dots, g(t) = t^{-2} + \dots$$

? E = ellinit([-1,1/4]); [f,g] = ellformaldifferential(E,7,'t); ? f %2 = 1 - 2*t^4 + 3/4*t^6 + 0(t^7) ? g %3 = t^-2 - t^2 + 1/2*t^4 + 0(t^5)

ellformalexp(E, serprec, n)

The elliptic formal exponential Exp attached to E is the isomorphism from the formal additive law to the formal group of E. It is normalized so as to be the inverse of the elliptic logarithm (see ellformallog): ExpoL = Id. Return n terms of this power series:

```
? E=ellinit([-1,1/4]); Exp = ellformalexp(E,10,'z)
%1 = z + 2/5*z^5 - 3/28*z^7 + 2/15*z^9 + 0(z^11)
? L = ellformallog(E,10,'t);
? subst(Exp,z,L)
%3 = t + 0(t^11)
```

ellformallog(E, serprec, n)

The formal elliptic logarithm is a series L in tK[[t]] such that $dL = \omega = dx/(2y + a_1x + a_3)$, the canonical invariant differential attached to the model E. It gives an isomorphism from the formal group of E to the additive formal group.

? E = ellinit([-1,1/4]); L = ellformallog(E, 9, 't) %1 = t - 2/5*t^5 + 3/28*t^7 + 2/3*t^9 + 0(t^10) ? [f,g] = ellformaldifferential(E,8,'t); ? L' - f %3 = 0(t^8)

ellformalpoint(E, serprec, n)

If E is an elliptic curve, return the coordinates x(t), y(t) in the formal group of the elliptic curve E in the formal parameter t = -x/y at oo:

$$x = t^{-2} - a_1 t^{-1} - a_2 - a_3 t + \dots$$
$$y = -t^{-3} - a_1 t^{-2} - a_2 t^{-1} - a_3 + \dots$$

Return *n* terms (seriesprecision by default) of these two power series, whose coefficients are in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6]$.

? E = ellinit([0,0,1,-1,0]); [x,y] = ellformalpoint(E,8,'t); ? x %2 = t^-2 - t + t^2 - t^4 + 2*t^5 + 0(t^6) ? y %3 = -t^-3 + 1 - t + t^3 - 2*t^4 + 0(t^5) ? E = ellinit([0,1/2]); ellformalpoint(E,7) %4 = $[x^-2 - 1/2*x^4 + 0(x^5), -x^-3 + 1/2*x^3 + 0(x^4)]$

ellformalw(E, serprec, n)

Return the formal power series w attached to the elliptic curve E, in the variable t:

$$w(t) = t^{3}(1 + a_{1}t + (a_{2} + a_{1}^{2})t^{2} + \dots + O(t^{n})).$$

which is the formal expansion of -1/y in the formal parameter t := -x/y at *oo* (take n = series precision if n is omitted). The coefficients of w belong to $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6]$.

? E=ellinit([3,2,-4,-2,5]); ellformalw(E, 5, 't) %1 = t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 0(t^8)

ellfromeqn(P)

Given a genus 1 plane curve, defined by the affine equation f(x, y) = 0, return the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of a Weierstrass equation for its Jacobian. This allows to recover a Weierstrass model for an elliptic curve given by a general plane cubic or by a binary quartic or biquadratic model. The function implements the $f : -- > f^*$ formulae of Artin, Tate and Villegas (Advances in Math. 198 (2005), pp. 366–382).

In the example below, the function is used to convert between twisted Edwards coordinates and Weierstrass coordinates.

? e = ellfromeqn(a*x^2+y^2 - (1+d*x^2*y^2)) %1 = [0, -a - d, 0, -4*d*a, 4*d*a^2 + 4*d^2*a] ? E = ellinit(ellfromeqn(y^2-x^2 - 1 +(121665/121666*x^2*y^2)),2^255-19); ? isprime(ellcard(E) / 8) %3 = 1

The elliptic curve attached to the sum of two cubes is given by

? ellfromeqn(x³+y³ - a) %1 = [0, 0, -9*a, 0, -27*a²]

Congruent number problem. Let *n* be an integer, if $a^2 + b^2 = c^2$ and ab = 2n, then by substituting *b* by 2n/a in the first equation, we get $((a^2 + (2n/a)^2) - c^2)a^2 = 0$. We set x = a, y = ac.

```
? En = ellfromeqn((x^2 + (2*n/x)^2 - (y/x)^2)*x^2)
%1 = [0, 0, 0, -16*n^2, 0]
```

For example 23 is congruent since the curve has a point of infinite order, namely:

```
? ellheegner( ellinit(subst(En, n, 23)) )
%2 = [168100/289, 68053440/4913]
```

ellfromj(j)

Returns the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of a fixed elliptic curve with *j*-invariant *j*.

ellgenerators(E)

If E is an elliptic curve over the rationals, return a \mathbb{Z} -basis of the free part of the Mordell-Weil group attached to E. This relies on the elldata database being installed and referencing the curve, and so is only available for curves over \mathbb{Z} of small conductors. If E is an elliptic curve over a finite field \mathbb{F}_q as output by ellinit, return a minimal set of generators for the group $E(\mathbb{F}_q)$.

Caution. When the group is not cyclic, of shape $\mathbb{Z}/d_1\mathbb{Z}x\mathbb{Z}/d_2\mathbb{Z}$ with $d_2||d_1$, the points [P,Q] returned by eligenerators need not have order d_1 and d_2 : it is true that P has order d_1 , but we only know that Q is a generator of $E(\mathbb{F}_q)/\langle P \rangle$ and that the Weil pairing w(P,Q) has order d_2 , see ??ellgroup. If you need generators [P,R] with R of order d_2 , find x such that R = Q - [x]P has order d_2 by solving the discrete logarithm problem $[d_2]Q = [x]([d_2]P)$ in a cyclic group of order d_1/d_2 . This will be very expensive if d_1/d_2 has a large prime factor.

ellglobalred(E)

Let *E* be an **ell** structure as output by **ellinit** attached to an elliptic curve defined over a number field. This function calculates the arithmetic conductor and the global Tamagawa number *c*. The result [N, v, c, F, L] is slightly different if *E* is defined over \mathbb{Q} (domain D = 1 in **ellinit**) or over a number field (domain *D* is a number field structure, including nfinit(x) representing \mathbb{Q} !):

- N is the arithmetic conductor of the curve,
- v is an obsolete field, left in place for backward compatibility. If E is defined over \mathbb{Q} , v gives the coordinate change for E to the standard minimal integral model (ellminimalmodel provides it in a cheaper way); if E is defined over another number field, v gives a coordinate change to an integral model (ellintegralmodel provides it in a cheaper way).
- c is the product of the local Tamagawa numbers c_p , a quantity which enters in the Birch and Swinnerton-Dyer conjecture,
- *F* is the factorization of *N*,
- *L* is a vector, whose *i*-th entry contains the local data at the *i*-th prime ideal divisor of *N*, i.e. L[i] = elllocalred(E, F[i, 1]). If *E* is defined over \mathbb{Q} , the local coordinate change has been deleted and replaced by a 0; if *E* is defined over another number field the local coordinate change to a local minimal model is given relative to the integral model afforded by *v* (so either start from an integral model so that *v* be trivial, or apply *v* first).

ellgroup(E, p, flag)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. We first describle the function when the field $K = \mathbb{F}_q$ is finite, it computes the structure of the finite abelian group $E(\mathbb{F}_q)$:

- if flag = 0, return the structure [] (trivial group) or $[d_1]$ (nontrivial cyclic group) or $[d_1, d_2]$ (noncyclic group) of $E(\mathbb{F}_q) \mathbb{Z}/d_1\mathbb{Z}x\mathbb{Z}/d_2\mathbb{Z}$, with $d_2 || d_1$.
- if flag = 1, return a triple [h, cyc, gen], where h is the curve cardinality, cyc gives the group structure as a product of cyclic groups (as per flag = 0). More precisely, if $d_2 > 1$, the output is $[d_1d_2, [d_1, d_2], [P, Q]]$ where P is of order d_1 and [P, Q] generates the curve. **Caution**. It is not guaranteed that Q has order d_2 , which in the worst case requires an expensive discrete log computation. Only that ellweilpairing (E, P, Q, d_1) has order d_2 .

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the structure of the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number ($K = \mathbb{Q}$) or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The curve is allowed to have bad reduction at p and in this case we consider the (cyclic) group of nonsingular points for the reduction of a minimal model at p.

If flag = 0, the equation not be minimal or even integral at p; of course, a minimal model will be more efficient.

If flag = 1, the requested generators depend on the model, which must then be minimal at p, otherwise an exception is thrown. Use ellintegralmodel and/or ellocalred first to reduce to this case.

```
? E = ellinit([0,1]); \\ y^2 = x^3 + 0.x + 1, defined over Q
? ellgroup(E, 7)
%2 = [6, 2] \\ Z/6 x Z/2, noncyclic
? E = ellinit([0,1] * Mod(1,11)); \\ defined over F_11
? ellgroup(E) \\ no need to repeat 11
\%4 = [12]
? ellgroup(E, 11) \\ ... but it also works
\%5 = [12]
? ellgroup(E, 13) \\ ouch, inconsistent input!
*** at top-level: ellgroup(E.13)
*** ellgroup: inconsistent moduli in Rg_to_Fp:
11
13
? ellgroup(E, 7, 1)
%6 = [12, [6, 2], [[Mod(2, 7), Mod(4, 7)], [Mod(4, 7), Mod(4, 7)]]]
```

Let us now consider curves of bad reduction, in this case we return the structure of the (cyclic) group of nonsingular points, satisfying $\#E_{ns}(\mathbb{F}_p) = p - a_p$:

```
? E = ellinit([0,5]);
? ellgroup(E, 5, 1)
%2 = [5, [5], [[Mod(4, 5), Mod(2, 5)]]]
? ellap(E, 5)
%3 = 0 \\ additive reduction at 5
? E = ellinit([0,-1,0,35,0]);
? ellgroup(E, 5, 1)
%5 = [4, [4], [[Mod(2, 5), Mod(2, 5)]]]
? ellap(E, 5)
%6 = 1 \\ split multiplicative reduction at 5
? ellgroup(E, 7, 1)
%7 = [8, [8], [[Mod(3, 7), Mod(5, 7)]]]
? ellap(E, 7)
%8 = -1 \\ nonsplit multiplicative reduction at 7
```

ellheegner(E)

Let E be an elliptic curve over the rationals, assumed to be of (analytic) rank 1. This returns a nontorsion rational point on the curve, whose canonical height is equal to the product of the elliptic regulator by the analytic Sha.

This uses the Heegner point method, described in Cohen GTM 239; the complexity is proportional to the product of the square root of the conductor and the height of the point (thus, it is preferable to apply it to strong Weil curves).

ellheight(E, P, Q, precision)

Let E be an elliptic curve defined over $K = \mathbb{Q}$ or a number field, as output by ellinit; it needs not be given by a minimal model although the computation will be faster if it is.

- Without arguments P, Q, returns the Faltings height of the curve E using Deligne normalization. For a rational curve, the normalization is such that the function returns -(1/2)*log(ellminimalmodel(E). area).
- If the argument $P \in E(K)$ is present, returns the global Néron-Tate height h(P) of the point, using the normalization in Cremona's Algorithms for modular elliptic curves.
- If the argument $Q \in E(K)$ is also present, computes the value of the bilinear form (h(P+Q)-h(P-Q))/4.

ellheightmatrix(E, x, precision)

x being a vector of points, this function outputs the Gram matrix of x with respect to the Néron-Tate height, in other words, the (i, j) component of the matrix is equal to ellbil(:math: `E,x[i],x[j])`. The rank of this matrix, at least in some approximate sense, gives the rank of the set of points, and if x is a basis of the Mordell-Weil group of E, its determinant is equal to the regulator of E. Note our height normalization follows Cremona's Algorithms for modular elliptic curves: this matrix should be divided by 2 to be in accordance with, e.g., Silverman's normalizations.

ellidentify(E)

Look up the elliptic curve E, defined by an arbitrary model over \mathbb{Q} , in the elldata database. Return [[N, M, G], C] where N is the curve name in Cremona's elliptic curve database, M is the minimal model, G is a \mathbb{Z} -basis of the free part of the Mordell-Weil group $E(\mathbb{Q})$ and C is the change of coordinates from E to M, suitable for ellchangecurve.

ellinit(x, D, precision)

Initialize an ell structure, attached to the elliptic curve E. E is either

• a 5-component vector $[a_1, a_2, a_3, a_4, a_6]$ defining the elliptic curve with Weierstrass equation

$$Y^2 + a_1 XY + a_3 Y = X^3 + a_2 X^2 + a_4 X + a_6,$$

• a 2-component vector $[a_4, a_6]$ defining the elliptic curve with short Weierstrass equation

$$Y^2 = X^3 + a_4 X + a_6,$$

• a character string in Cremona's notation, e.g. "11a1", in which case the curve is retrieved from the elldata database if available.

The optional argument D describes the domain over which the curve is defined:

- the t_INT 1 (default): the field of rational numbers Q.
- a t_INT p, where p is a prime number: the prime finite field \mathbb{F}_p .
- an t_INTMOD Mod(a, p), where p is a prime number: the prime finite field \mathbb{F}_p .
- a t_FFELT, as returned by ffgen: the corresponding finite field \mathbb{F}_q .
- a t_PADIC, O(pⁿ): the field Q_p, where p-adic quantities will be computed to a relative accuracy of n digits. We advise to input a model defined over Q for such curves. In any case, if you input an approximate model with t_PADIC coefficients, it will be replaced by a lift to Q (an exact model "close" to the one that was input) and all quantities will then be computed in terms of this lifted model, at the given accuracy.
- a t_REAL x: the field \mathbb{C} of complex numbers, where floating point quantities are by default computed to a relative accuracy of precision(x). If no such argument is given, the value of realprecision at the time ellinit is called will be used.
- a number field K, given by a nf or bnf structure; a bnf is required for ellminimalmodel.
- a prime ideal p, given by a prid structure; valid if x is a curve defined over a number field K and the equation is integral and minimal at p.

This argument D is indicative: the curve coefficients are checked for compatibility, possibly changing D; for instance if D = 1 and an t_INTMOD is found. If inconsistencies are detected, an error is raised:

If the curve coefficients are too general to fit any of the above domain categories, only basic operations, such as point addition, will be supported later.

If the curve (seen over the domain D) is singular, fail and return an empty vector [].

```
? E = ellinit([0,0,0,0,1]); \\ y^2 = x^3 + 1, over Q
? E = ellinit([0,1]); \\ the same curve, short form
? E = ellinit("36a1"); \\ sill the same curve, Cremona's notations
? E = ellinit([0,1], 2) \\ over F2: singular curve
```

%4 = []
? E = ellinit(['a4,'a6] * Mod(1,5)); \\ over F_5[a4,a6], basic support !

The result of ellinit is an *ell* structure. It contains at least the following information in its components:

```
a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j.
```

All are accessible via member functions. In particular, the discriminant is :math:`E.disc`, and the *j*-invariant is :math:`E.j`.

? E = ellinit([a4, a6]); ? E.disc %2 = -64*a4^3 - 432*a6^2 ? E.j %3 = -6912*a4^3/(-4*a4^3 - 27*a6^2)

Further components contain domain-specific data, which are in general dynamic: only computed when needed, and then cached in the structure.

```
? E = ellinit([2,3], 10^60+7); \\ E over F_p, p large
? ellap(E)
time = 4,440 ms.
%2 = -1376268269510579884904540406082
? ellcard(E); \\ now instantaneous !
time = 0 ms.
? ellgenerators(E);
time = 5,965 ms.
? ellgenerators(E); \\ second time instantaneous
time = 0 ms.
```

See the description of member functions related to elliptic curves at the beginning of this section.

ellintegralmodel(E, v)

Let *E* be an ell structure over a number field *K* or \mathbb{Q}_p . This function returns an integral model. If *v* is present, sets v = [u, 0, 0, 0] to the corresponding change of variable: the return value is identical to that of ellchangecurve(E, v).

```
? e = ellinit([1/17,1/42]);
? e = ellintegralmodel(e,&v);
? e[1..5]
%3 = [0, 0, 0, 15287762448, 3154568630095008]
? v
%4 = [1/714, 0, 0, 0]
```

ellisdivisible(E, P, n, Q)

Given E/K a number field and P in E(K) return 1 if P = [n]R for some R in E(K) and set Q to one such R; and return 0 otherwise. The integer $n \ge 0$ may be given as ellxn(E,n), if many points need to be tested.

? K = nfinit(polcyclo(11,t)); ? E = ellinit([0,-1,1,0,0], K); ? P = [0,0]; ? ellorder(E,P) %4 = 5

```
? ellisdivisible(E,P,5, &Q)
%5 = 1
? lift(Q)
%6 = [-t^7-t^6-t^5-t^4+1, -t^9-2*t^8-2*t^7-3*t^6-3*t^5-2*t^4-2*t^3-t^2-1]
? ellorder(E, Q)
%7 = 25
```

The algebraic complexity of the underlying algorithm is in $O(n^4)$, so it is advisable to first factor n, then use a chain of checks attached to the prime divisors of n: the function will do it itself unless n is given in ellxn form.

ellisogeny(E, G, only_image, x, y)

Given an elliptic curve E, a finite subgroup G of E is given either as a generating point P (for a cyclic G) or as a polynomial whose roots vanish on the x-coordinates of the nonzero elements of G (general case and more efficient if available). This function returns the $[a_1, a_2, a_3, a_4, a_6]$ invariants of the quotient elliptic curve E/Gand (if only_image is zero (the default)) a vector of rational functions [f, g, h] such that the isogeny $E \to E/G$ is given by $(x, y) : -- > (f(x)/h(x)^2, g(x, y)/h(x)^3)$.

```
? E = ellinit([0,1]);
? elltors(E)
%2 = [6, [6], [[2, 3]]]
? ellisogeny(E, [2,3], 1) \\ Weierstrass model for E/<P>
%3 = [0, 0, 0, -135, -594]
? ellisogeny(E,[-1,0])
%4 = [[0,0,0,-15,22], [x^3+2*x^2+4*x+3, y*x^3+3*y*x^2-2*y, x+1]]
```

ellisogenyapply(f, g)

Given an isogeny of elliptic curves $f: E' \to E$ (being the result of a call to ellisogeny), apply f to g:

- if g is a point P in the domain of f, return the image f(P);
- if $g: E^{"} \to E'$ is a compatible isogeny, return the composite isogeny $fog: E^{"} \to E$.

```
? one = ffgen(101, 't)^0;
? E = ellinit([6, 53, 85, 32, 34] * one);
? P = [84, 71] * one;
? ellorder(E, P)
%4 = 5
? [F, f] = ellisogeny(E, P); \\ f: E->F = E/<P>
? ellisogenyapply(f, P)
%6 = [0]
? F = ellinit(F);
? Q = [89, 44] * one;
? ellorder(F, Q)
%9 = 2
? [G, g] = ellisogeny(F, Q); \\ g: F->G = F/<Q>
? gof = ellisogenyapply(g, f); \\ gof: E -> G
```

ellisomat(E, p, fl)

Given an elliptic curve E defined over a number field K, compute representatives of the isomorphism classes of elliptic curves defined over K and K-isogenous to E. We assume that E does not have CM over K (otherwise that set would be infinite). For any such curve E_i , let $f_i : E \to E_i$ be a rational isogeny of minimal degree and let $g_i : E_i \to E$ be the dual isogeny; and let M be the matrix such that $M_{i,j}$ is the minimal degree for an isogeny $E_i \to E_j$.

The function returns a vector [L, M] where L is a list of triples $[E_i, f_i, g_i]$ (flag = 0), or simply the list of E_i (flag = 1, which saves time). The curves E_i are given in $[a_4, a_6]$ form and the first curve E_1 is isomorphic to E by f_1 .

If p is set, it must be a prime number; in this which case only isogenies of degree a power of p are considered.

Over a number field, the possible isogeny degrees are determined by Billerey algorithm.

```
? E = ellinit("14a1");
? [L,M] = ellisomat(E);
? LE = apply(x->x[1], L) \setminus list of curves
%3 = [[215/48, -5291/864], [-675/16, 6831/32], [-8185/48, -742643/864],
[-1705/48,-57707/864],[-13635/16,306207/32],[-131065/48,-47449331/864]]
? L[2][2] \\ isogeny f_2
\%4 = [x^3+3/4*x^2+19/2*x-311/12],
1/2*x^{4}+(y+1)*x^{3}+(y-4)*x^{2}+(-9*y+23)*x+(55*y+55/2),x+1/3
? L[2][3] \setminus dual isogeny g_2
\%5 = [1/9*x^3-1/4*x^2-141/16*x+5613/64]
-1/18*x^{4}+(1/27*y-1/3)*x^{3}+(-1/12*y+87/16)*x^{2}+(49/16*y-48)*x
+(-3601/64*y+16947/512),x-3/4]
? apply(E->ellidentify(ellinit(E))[1][1], LE)
%6 = ["14a1","14a4","14a3","14a2","14a6","14a5"]
? M
%7 =
[1 3 3 2 6 6]
[3 1 9 6 2 18]
[3 9 1 6 18 2]
[2 6 6 1 3 3]
[6 2 18 3 1 9]
[6 18 2 3 9 1]
```

ellisoncurve(E, z)

Gives 1 (i.e. true) if the point z is on the elliptic curve E, 0 otherwise. If E or z have imprecise coefficients, an attempt is made to take this into account, i.e. an imprecise equality is checked, not a precise one. It is allowed for z to be a vector of points in which case a vector (of the same type) is returned.

ellisotree(E)

Given an elliptic curve E defined over \mathbb{Q} or a set of \mathbb{Q} -isogenous curves as given by ellisomat, return a pair [L, M] where

- L lists the minimal models of the isomorphism classes of elliptic curves \mathbb{Q} -isogenous to E (or in the set of isogenous curves),
- M is the adjacency matrix of the prime degree isogenies tree: there is an edge from E_i to E_j if there is an isogeny E_i → E_j of prime degree such that the Néron differential forms are preserved.

```
? E = ellinit("14a1");
? [L,M] = ellisotree(E);
? M
%3 =
[0 0 3 2 0 0]
```

```
[3 0 0 0 2 0]
[0 0 0 0 0 2]
[0 0 0 0 0 3]
[0 0 0 3 0 0]
[0 0 0 0 0 0]
? [L2,M2] = ellisotree(ellisomat(E,2,1));
%4 =
[0 2]
[0 0]
? [L3,M3] = ellisotree(ellisomat(E,3,1));
? M3
%6 =
[0 0 3]
[3 0 0]
[0 0 0]
```

Compare with the result of ellisomat.

```
? [L,M]=ellisomat(E,,1);
? M
%7 =
[1 3 3 2 6 6]
[3 1 9 6 2 18]
[3 9 1 6 18 2]
[2 6 6 1 3 3]
[6 2 18 3 1 9]
[6 18 2 3 9 1]
```

ellissupersingular(E, p)

Return 1 if the elliptic curve E defined over a number field, \mathbb{Q}_p or a finite field is supersingular at p, and 0 otherwise. If the curve is defined over a number field, p must be explicitly given, and must be a prime number, resp. a maximal ideal, if the curve is defined over \mathbb{Q} , resp. a general number field: we return 1 if and only if E has supersingular good reduction at p.

Alternatively, E can be given by its j-invariant in a finite field. In this case p must be omitted.

```
? setrand(1); \\ make the choice of g deterministic
? g = ffprimroot(ffgen(7^5))
%1 = 4*x^4 + 5*x^3 + 6*x^2 + 5*x + 6
? [g^n | n <- [1 .. 7^5 - 1], ellissupersingular(g^n)]</pre>
```

```
%2 = [6]
? K = nfinit(y^3-2); P = idealprimedec(K, 2)[1];
? E = ellinit([y,1], K);
? ellissupersingular(E, P)
%5 = 1
? Q = idealprimedec(K,5)[1];
? ellissupersingular(E, Q)
%6 = 0
```

ellj(*x*, *precision*)

Elliptic j-invariant. x must be a complex number with positive imaginary part, or convertible into a power series or a p-adic number with positive valuation.

elllocalred(E, p)

Calculates the Kodaira type of the local fiber of the elliptic curve E at p. E must be an ell structure as output by ellinit, over \mathbb{Q}_{ℓ} (p better left omitted, else equal to ℓ) over \mathbb{Q} (p a rational prime) or a number field K(p a maximal ideal given by a prid structure). The result is a 4-component vector [f, kod, v, c]. Here f is the exponent of p in the arithmetic conductor of E, and kod is the Kodaira type which is coded as follows:

1 means good reduction (type I:math:_0), 2, 3 and 4 mean types II, III and IV respectively, $4+\nu$ with $\nu > 0$ means type I:math:_nu; finally the opposite values -1, -2, etc. refer to the starred types I:math:_ 0^* , II:math:*, etc. The third component v is itself a vector [u, r, s, t] giving the coordinate changes done during the local reduction; u = 1 if and only if the given equation was already minimal at p. Finally, the last component c is the local Tamagawa number c_p .

elllog(E, P, G, o)

Given two points P and G on the elliptic curve E/\mathbb{F}_q , returns the discrete logarithm of P in base G, i.e. the smallest nonnegative integer n such that P = [n]G. See znlog for the limitations of the underlying discrete log algorithms. If present, o represents the order of G, see DLfun (in the PARI manual); the preferred format for this parameter is [N, factor(N)], where N is the order of G.

If no o is given, assume that G generates the curve. The function also assumes that P is a multiple of G.

```
? a = ffgen(ffinit(2,8),'a);
? E = ellinit([a,1,0,0,1]); \\ over F_{2^8}
? x = a^3; y = ellordinate(E,x)[1];
? P = [x,y]; G = ellmul(E, P, 113);
? ord = [242, factor(242)]; \\ P generates a group of order 242. Initialize.
? ellorder(E, G, ord)
%4 = 242
? e = elllog(E, P, G, ord)
%5 = 15
? ellmul(E,G,e) == P
%6 = 1
```

elllseries(E, s, A, precision)

This function is deprecated, use lfun(E,s) instead.

E being an elliptic curve, given by an arbitrary model over \mathbb{Q} as output by ellipsit, this function computes the value of the *L*-series of *E* at the (complex) point *s*. This function uses an $O(N^{1/2})$ algorithm, where *N* is the conductor.

The optional parameter A fixes a cutoff point for the integral and is best left omitted; the result must be independent of A, up to realprecision, so this allows to check the function's accuracy.

ellminimaldisc(E)

E being an elliptic curve defined over a number field output by ellinit, return the minimal discriminant ideal of E.

ellminimalmodel(E, v)

Let *E* be an ell structure over a number field *K*. This function determines whether *E* admits a global minimal integral model. If so, it returns it and sets v = [u, r, s, t] to the corresponding change of variable: the return value is identical to that of ellchangecurve(E, v).

Else return the (nonprincipal) Weierstrass class of E, i.e. the class of $\prod p^{(v_p \Delta - \delta_p)/12}$ where $\Delta = E.disc$ is the model's discriminant and p_p^{δ} is the local minimal discriminant. This function requires either that E be defined over the rational field \mathbb{Q} (with domain D = 1 in ellinit), in which case a global minimal model always exists, or over a number field given by a *bnf* structure. The Weierstrass class is given in bnfisprincipal format, i.e. in terms of the K.gen generators.

The resulting model has integral coefficients and is everywhere minimal, the coefficients a_1 and a_3 are reduced modulo 2 (in terms of the fixed integral basis K.zk) and a_2 is reduced modulo 3. Over \mathbb{Q} , we further require that a_1 and a_3 be 0 or 1, that a_2 be 0 or 1 and that u > 0 in the change of variable: both the model and the change of variable v are then unique.

```
? e = ellinit([6,6,12,55,233]); \\ over Q
? E = ellminimalmodel(e, &v);
? E[1..5]
%3 = [0, 0, 0, 1, 1]
? v
%4 = [2, -5, -3, 9]
```

```
? K = bnfinit(a^2-65); \\ over a nonprincipal number field
? K.cyc
%2 = [2]
? u = Mod(8+a, K.pol);
? E = ellinit([1,40*u+1,0,25*u^2,0], K);
? ellminimalmodel(E) \\ no global minimal model exists over Z_K
%6 = [1]~
```

ellminimaltwist(E, flag)

Let *E* be an elliptic curve defined over \mathbb{Q} , return a discriminant *D* such that the twist of *E* by *D* is minimal among all possible quadratic twists, i.e. if flag = 0, its minimal model has minimal discriminant, or if flag = 1, it has minimal conductor.

In the example below, we find a curve with j-invariant 3 and minimal conductor.

```
? E = ellminimalmodel(ellinit(ellfromj(3)));
? ellglobalred(E)[1]
%2 = 357075
? D = ellminimaltwist(E,1)
%3 = -15
? E2 = ellminimalmodel(ellinit(elltwist(E,D)));
? ellglobalred(E2)[1]
%5 = 14283
```

In the example below, flag = 0 and flag = 1 give different results.

```
? E = ellinit([1,0]);
? D0 = ellminimaltwist(E,0)
```

```
%7 = 1
? D1 = ellminimaltwist(E,1)
%8 = 8
? E0 = ellminimalmodel(ellinit(elltwist(E,D0)));
? [E0.disc, ellglobalred(E0)[1]]
%10 = [-64, 64]
? E1 = ellminimalmodel(ellinit(elltwist(E,D1)));
? [E1.disc, ellglobalred(E1)[1]]
%12 = [-4096, 32]
```

ellmoddegree(e)

e being an elliptic curve defined over \mathbb{Q} output by ellipit, compute the modular degree of *e* divided by the square of the Manin constant *c*. It is conjectured that c = 1 for the strong Weil curve in the isogeny class (optimal quotient of $J_0(N)$) and this can be proven using ellweilcurve when the conductor *N* is moderate.

```
? E = ellinit("11a1"); \\ from Cremona table: strong Weil curve and c = 1
? [v,smith] = ellweilcurve(E); smith \\ proof of the above
%2 = [[1, 1], [5, 1], [1, 1/5]]
? ellmoddegree(E)
%3 = 1
? [ellidentify(e)[1][1] | e<-v]
%4 = ["11a1", "11a2", "11a3"]
? ellmoddegree(ellinit("11a3"))
%5 = 5
? ellmoddegree(ellinit("11a3"))
%6 = 1/5
```

The modular degree of 11a1 is 1 (because ellweilcurve or Cremona's table prove that the Manin constant is 1 for this curve); the output of ellweilcurve also proves that the Manin constants of 11a2 and 11a3 are 1 and 5 respectively, so the actual modular degree of both 11a2 and 11a3 is 5.

ellmodulareqn(N, x, y)

Given a prime N < 500, return a vector [P, t] where P(x, y) is a modular equation of level N, i.e. a bivariate polynomial with integer coefficients; t indicates the type of this equation: either *canonical* (t = 0) or *Atkin* (t = 1). This function requires the seadata package and its only use is to give access to the package contents. See polmodular for a more general and more flexible function.

Let j be the j-invariant function. The polynomial P satisfies the functional equation,

$$P(f, j) = P(f || W_N, j || W_N) = 0$$

for some modular function $f = f_N$ (hand-picked for each fixed N to minimize its size, see below), where $W_N(\tau) = -1/(N\tau)$ is the Atkin-Lehner involution. These two equations allow to compute the values of the classical modular polynomial Φ_N , such that $\Phi_N(j(\tau), j(N\tau)) = 0$, while being much smaller than the latter. More precisely, we have $j(W_N(\tau)) = j(N\tau)$; the function f is invariant under $\Gamma_0(N)$ and also satisfies

- for Atkin type: $f || W_N = f$;
- for canonical type: let s = 12/gcd(12, N-1), then $f || W_N = N^s/f$. In this case, f has a simple definition: $f(\tau) = N^s(\eta(N\tau)/\eta(\tau))^{2s}$, where η is Dedekind's eta function.

The following GP function returns values of the classical modular polynomial by eliminating $f_N(\tau)$ in the above functional equation, for $N \le 31$ or $N \in 41, 47, 59, 71$.
```
classicaleqn(N, X='X, Y='Y)=
{
    my([P,t] = ellmodulareqn(N), Q, d);
    if (poldegree(P,'y) > 2, error("level unavailable in classicaleqn"));
    if (t == 0, \\ Canonical
    my(s = 12/gcd(12,N-1));
    Q = 'x^(N+1) * substvec(P,['x,'y],[N^s/'x,Y]);
    d = N^(s*(2*N+1)) * (-1)^(N+1);
    , \\ Atkin
    Q = subst(P,'y,Y);
    d = (X-Y)^(N+1));
    polresultant(subst(P,'y,X), Q) / d;
}
```

ellmul(*E*, *z*, *n*)

Computes [n]z, where z is a point on the elliptic curve E. The exponent n is in \mathbb{Z} , or may be a complex quadratic integer if the curve E has complex multiplication by n (if not, an error message is issued).

The simple-minded algorithm for the CM case assumes that we are in characteristic 0, and that the quadratic order to which n belongs has small discriminant.

ellneg(E, z)

Opposite of the point z on elliptic curve E.

ellnonsingularmultiple(E, P)

Given an elliptic curve E/\mathbb{Q} (more precisely, a model defined over \mathbb{Q} of a curve) and a rational point $P \in E(\mathbb{Q})$, returns the pair [R, n], where n is the least positive integer such that R := [n]P has good reduction at every prime. More precisely, its image in a minimal model is everywhere nonsingular.

```
? e = ellinit("57a1"); P = [2,-2];
? ellnonsingularmultiple(e, P)
%2 = [[1, -1], 2]
? e = ellinit("396b2"); P = [35, -198];
? [R,n] = ellnonsingularmultiple(e, P);
? n
%5 = 12
```

ellorder(E, z, o)

Gives the order of the point z on the elliptic curve E, defined over a finite field or a number field. Return (the

impossible value) zero if the point has infinite order.

```
? E = ellinit([-157^2,0]); \setminus the "157-is-congruent" curve
? P = [2,2]; ellorder(E, P)
%2 = 2
? P = ellheegner(E); ellorder(E, P) \\ infinite order
\%3 = 0
? K = nfinit(polcyclo(11,t)); E=ellinit("11a3", K); T = elltors(E);
? ellorder(E, T.gen[1])
\%5 = 25
? E = ellinit(ellfromj(ffgen(5^10)));
? ellcard(E)
%7 = 9762580
? P = random(E); ellorder(E, P)
\%8 = 4881290
? p = 2^{160+7}; E = ellinit([1,2], p);
? N = ellcard(E)
%9 = 1461501637330902918203686560289225285992592471152
? o = [N, factor(N)];
? for(i=1,100, ellorder(E,random(E)))
time = 260 \text{ ms}.
```

The parameter o, is now mostly useless, and kept for backward compatibility. If present, it represents a nonzero multiple of the order of z, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the cardinality of the curve. It is no longer needed since PARI is now able to compute it over large finite fields (was restricted to small prime fields at the time this feature was introduced), and caches the result in E so that it is computed and factored only once. Modifying the last example, we see that including this extra parameter provides no improvement:

? o = [N, factor(N)]; ? for(i=1,100, ellorder(E,random(E),o)) time = 260 ms.

ellordinate(E, x, precision)

Gives a 0, 1 or 2-component vector containing the y-coordinates of the points of the curve E having x as x-coordinate.

ellpadicL(E, p, n, s, r, D)

Returns the value (or r-th derivative) on a character χ^s of \mathbb{Z}_p^* of the p-adic L-function of the elliptic curve E/\mathbb{Q} , twisted by D, given modulo p^n .

Characters. The set of continuous characters of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$ is identified to \mathbb{Z}_p^* via the cyclotomic character χ with values in $\overline{\mathbb{Q}_p}^*$. Denote by $\tau : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ the Teichmüller character, with values in the (p-1)-th roots of 1 for p! = 2, and -1, 1 for p = 2; finally, let $\langle \chi \rangle = \chi \tau^{-1}$, with values in $1 + 2p\mathbb{Z}_p$. In GP, the continuous character of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$ given by $\langle \chi \rangle^{s_1} \tau^{s_2}$ is represented by the pair of integers $s = (s_1, s_2)$, with $s_1 \in \mathbb{Z}_p$ and $s_2 modp - 1$ for p > 2, (resp. mod 2 for p = 2); s may be also an integer, representing (s, s) or χ^s .

The :math:`p-adic L function.` The p-adic L function L_p is defined on the set of continuous characters of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$, as $\int_{\mathbb{Z}^*} \chi^s d\mu$ for a certain p-adic distribution μ on \mathbb{Z}_p^* . The derivative is given by

$$L_p^{(r)}(E,\chi^s) = \int_{\mathbb{Z}_p^s} \log_p^r(a) \chi^s(a) d\mu(a).$$

More precisely:

• When E has good supersingular reduction, L_p takes its values in $D := H^1_{dR}(E/\mathbb{Q}) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ and satisfies

where: math: `F` is the Frobenius,: math: `L(E,1)` is the value of the complex: math: `L` function at: math: `1`,: math: `1`,:

The function returns the components of $L_p^{(r)}(E, \chi^s)$ in the basis $(\omega, F\omega)$.

• When E has ordinary good reduction, this method only defines the projection of $L_p(E, \chi^s)$ on the α eigenspace, where α is the unit eigenvalue for F. This is what the function returns. We have

$$(1 - \alpha^{-1})^{-2} L_{p,\alpha}(E, \chi^0) = L(E, 1) / \Omega.$$

Two supersingular examples:

? cxL(e) = bestappr(ellL1(e) / e.omega[1]); ? e = ellinit("17a1"); p=3; \\ supersingular, a3 = 0 ? L = ellpadicL(e,p,4); ? F = [0,-p;1,ellap(e,p)]; \\ Frobenius matrix in the basis (omega,F(omega)) ? (1-p^(-1)*F)^-2 * L / cxL(e) %5 = [1 + 0(3^5), 0(3^5)]~ \\ [1,0]~ ? e = ellinit("116a1"); p=3; \\ supersingular, a3 != 0~ ? L = ellpadicL(e,p,4); ? F = [0,-p; 1,ellap(e,p)]; ? (1-p^(-1)*F)^-2*L~ / cxL(e) %9 = [1 + 0(3^4), 0(3^5)]~

Good ordinary reduction:

```
? e = ellinit("17a1"); p=5; ap = ellap(e,p)
%1 = -2 \\ ordinary
? L = ellpadicL(e,p,4)
%2 = 4 + 3*5 + 4*5^2 + 2*5^3 + 0(5^4)
? al = padicappr(x^2 - ap*x + p, ap + 0(p^7))[1];
? (1-al^(-1))^(-2) * L / cxL(e)
%4 = 1 + 0(5^4)
```

Twist and Teichmüller:

? e = ellinit("17a1"); p=5; \\ ordinary \\ 2nd derivative at tau^1, twist by -7 ? ellpadicL(e, p, 4, [0,1], 2, -7) %2 = 2*5^2 + 5^3 + O(5^4)

We give an example of non split multiplicative reduction (see ellpadicbsd for more examples).

```
? e=ellinit("15a1"); p=3; n=5;
? L = ellpadicL(e,p,n)
%2 = 2 + 3 + 3^2 + 3^3 + 3^4 + 0(3^5)
? (1 - ellap(e,p))^(-1) * L / cxL(e)
%3 = 1 + 0(3^5)
```

This function is a special case of mspadicL and it also appears as the first term of mspadicseries:

```
? e = ellinit("17a1"); p=5;
? L = ellpadicL(e,p,4)
```

```
%2 = 4 + 3*5 + 4*5^2 + 2*5^3 + 0(5^4)
? [M,phi] = msfromell(e, 1);
? Mp = mspadicinit(M, p, 4);
? mu = mspadicmoments(Mp, phi);
? mspadicL(mu)
%6 = 4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 0(5^6)
? mspadicseries(mu)
%7 = (4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 0(5^6))
+ (3 + 3*5 + 5^2 + 5^3 + 0(5^4))*x
+ (2 + 3*5 + 5^2 + 0(5^3))*x^2
+ (3 + 4*5 + 4*5^2 + 0(5^3))*x^3
+ (3 + 2*5 + 0(5^2))*x^4 + 0(x^5)
```

These are more cumbersome than ellpadicL but allow to compute at different characters, or successive derivatives, or to twist by a quadratic character essentially for the cost of a single call to ellpadicL due to precomputations.

ellpadicbsd(E, p, n, D)

Given an elliptic curve E over \mathbb{Q} , its quadratic twist E_D and a prime number p, this function is a p-adic analog of the complex functions ellanalyticrank and ellbsd. It calls ellpadicL with initial accuracy p^n and may increase it internally; it returns a vector $[r, L_p]$ where

- L_p is a *p*-adic number (resp. a pair of *p*-adic numbers if *E* has good supersingular reduction) defined modulo p^N , conjecturally equal to R_pS , where R_p is the *p*-adic regulator as given by ellpadicregulator (in the basis $(\omega, F\omega)$) and *S* is the cardinal of the Tate-Shafarevich group for the quadratic twist E_D .
- r is an upper bound for the analytic rank of the p-adic L-function attached to E_D : we know for sure that the *i*-th derivative of $L_p(E_D, .)$ at χ^0 is $O(p^N)$ for all i < r and that its r-th derivative is nonzero; it is expected that the true analytic rank is equal to the rank of the Mordell-Weil group $E_D(\mathbb{Q})$, plus 1 if the reduction of E_D at p is split multiplicative; if r = 0, then both the analytic rank and the Mordell-Weil rank are unconditionnally 0.

Recall that the *p*-adic BSD conjecture (Mazur, Tate, Teitelbaum, Bernardi, Perrin-Riou) predicts an explicit link between R_pS and

$$(1-p^{-1}F)^{-2}.L_p^{(r)}(E_D,\chi^0)/r!$$

where r is the analytic rank of the p-adic L-function attached to E_D and F is the Frobenius on H_{dR}^1 ; see ellpadicL for definitions.

```
? E = ellinit("11a1"); p = 7; n = 5; \\ good ordinary
? ellpadicbsd(E, 7, 5) \\ rank 0,
%2 = [0, 1 + 0(7^5)]
? E = ellinit("91a1"); p = 7; n = 5; \\ non split multiplicative
? [r,Lp] = ellpadicbsd(E, p, n)
%5 = [1, 2*7 + 6*7^2 + 3*7^3 + 7^4 + 0(7^5)]
? R = ellpadicregulator(E, p, n, E.gen)
%6 = 2*7 + 6*7^2 + 3*7^3 + 7^4 + 5*7^5 + 0(7^6)
? sha = Lp/R
%7 = 1 + 0(7^4)
? E = ellinit("91b1"); p = 7; n = 5; \\ split multiplicative
? [r,Lp] = ellpadicbsd(E, p, n)
```

```
\%9 = [2, 2*7 + 7^2 + 5*7^3 + 0(7^4)]
? ellpadicregulator(E, p, n, E.gen)
\%10 = 2*7 + 7^{2} + 5*7^{3} + 6*7^{4} + 2*7^{5} + 0(7^{6})
? [rC, LC] = ellanalyticrank(E);
? [r, rC]
%12 = [2, 1] \setminus r = rC+1 because of split multiplicative reduction
? E = ellinit("53a1"); p = 5; n = 5; \setminus supersingular
? [r, Lp] = ellpadicbsd(E, p, n);
? r
\%15 = 1
? Lp
\%16 = [3*5 + 2*5^2 + 2*5^5 + 0(5^6), \
5 + 3*5^2 + 4*5^3 + 2*5^4 + 5^5 + 0(5^6)
? R = ellpadicregulator(E, p, n, E.gen)
17 = [3*5 + 2*5^2 + 2*5^5 + 0(5^6), 5 + 3*5^2 + 4*5^3 + 2*5^4 + 0(5^5)]
\ expect Lp = R*#Sha, hence (conjecturally) #Sha = 1
? E = ellinit("84a1"); p = 11; n = 6; D = -443;
? [r,Lp] = ellpadicbsd(E, 11, 6, D) \\ Mordell-Weil rank 0, no regulator
%19 = [0, 3 + 2*11 + 0(11^6)]
? lift(Lp) \setminus expected cardinal for Sha is 5^2
\%20 = 25
? ellpadicbsd(E, 3, 12, D) \\ at 3
\%21 = [1, 1 + 2*3 + 2*3^2 + 0(3^8)]
? ellpadicbsd(E, 7, 8, D) \setminus and at 7
\%22 = [0, 4 + 3*7 + 0(7*8)]
```

ellpadicfrobenius(E, p, n)

If p > 2 is a prime and E is an elliptic curve on \mathbb{Q} with good reduction at p, return the matrix of the Frobenius endomorphism φ on the crystalline module $D_p(E) = \mathbb{Q}_p \otimes H^1_{dR}(E/\mathbb{Q})$ with respect to the basis of the given model $(\omega, \eta = x\omega)$, where $\omega = dx/(2y + a_1x + a_3)$ is the invariant differential. The characteristic polynomial of φ is $x^2 - a_p x + p$. The matrix is computed to absolute p-adic precision p^n .

```
? E = ellinit([1,-1,1,0,0]);
? F = ellpadicfrobenius(E,5,3);
? lift(F)
%3 =
[120 29]
[ 55 5]
? charpoly(F)
%4 = x^2 + 0(5^3)*x + (5 + 0(5^3))
? ellap(E, 5)
%5 = 0
```

ellpadicheight(E, p, n, P, Q)

Cyclotomic *p*-adic height of the rational point *P* on the elliptic curve *E* (defined over \mathbb{Q}), given to *n p*-adic digits. If the argument *Q* is present, computes the value of the bilinear form (h(P+Q) - h(P-Q))/4.

Let $D := H^1_{dR}(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ be the \mathbb{Q}_p vector space spanned by ω (invariant differential $dx/(2y+a_1x+a_3)$ related to the given model) and $\eta = x\omega$. Then the cyclotomic *p*-adic height h_E associates to $P \in E(\mathbb{Q})$ an element $f\omega + g\eta$ in *D*. This routine returns the vector [f, g] to *n p*-adic digits. If $P \in E(\mathbb{Q})$ is in the kernel of reduction mod p and if its reduction at all finite places is non singular, then $g = -(\log_E P)^2$, where \log_E is the logarithm for the formal group of E at p.

If furthermore the model is of the form $Y^2 = X^3 + aX + b$ and P = (x, y), then

 $f = \log_p(denominator(x)) - 2\log_p(\sigma(P))$

where $\sigma(P)$ is given by ellsigma(E, P).

Recall (Advanced topics in the arithmetic of elliptic curves, Theorem 3.2) that the local height function over the complex numbers is of the form

$$\lambda(z) = -\log(||E.disc||)/6 + \Re(z\eta(z)) - 2\log(\sigma(z)).$$

(N.B. our normalization for local and global heights is twice that of Silverman's).

```
? E = ellinit([1,-1,1,0,0]); P = [0,0];
? ellpadicheight(E,5,3, P)
%2 = [3*5 + 5^2 + 2*5^3 + 0(5^4), 5^2 + 4*5^4 + 0(5^5)]
? E = ellinit("11a1"); P = [5,5]; \\ torsion point
? ellpadicheight(E,19,6, P)
%4 = [0, 0]
? E = ellinit([0,0,1,-4,2]); P = [-2,1];
? ellpadicheight(E,3,3, P)
%6 = [2*3^2 + 2*3^3 + 3^4 + 0(3^5), 2*3^2 + 3^4 + 0(3^5)]
? ellpadicheight(E,3,5, P, elladd(E,P,P))
%7 = [3^2 + 2*3^3 + 0(3^7), 3^2 + 3^3 + 2*3^4 + 3^5 + 0(3^7)]
```

• When E has good ordinary reduction at p or non split multiplicative reduction, the "canonical" p-adic height is given by

s2 = ellpadics2(E,p,n); ellpadicheight(E, p, n, P) * [1,-s2]~

Since s_2 does not depend on P, it is preferable to compute it only once:

```
? E = ellinit("5077a1"); p = 5; n = 7; \\ rank 3
? s2 = ellpadics2(E,p,n);
? M = ellpadicheightmatrix(E,p, n, E.gen) * [1,-s2]~;
? matdet(M) \\ p-adic regulator on the points in E.gen
%4 = 5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 0(5^7)
```

• When E has split multiplicative reduction at p (Tate curve), the "canonical" p-adic height is given by

```
Ep = ellinit(E[1..5], O(p^(n))); \setminus E seen as a Tate curve over Qp [u2,u,q] = Ep.tate;
ellpadicheight(E, p, n, P) * [1,-s2 + 1/log(q)/u2]]\sim
```

where s_2 is as above. For example,

```
? E = ellinit("91b1"); P =[-1, 3]; p = 7; n = 5;
? Ep = ellinit(E[1..5], 0(p^(n)));
? s2 = ellpadics2(E,p,n);
? [u2,u,q] = Ep.tate;
```

? H = ellpadicheight(E,p, n, P) * $[1,-s2 + 1/\log(q)/u2] \sim$ %5 = 2*7 + 7^2 + 5*7^3 + 6*7^4 + 2*7^5 + 0(7^6)

These normalizations are chosen so that *p*-adic BSD conjectures are easy to state, see ellpadicbsd.

ellpadicheightmatrix(E, p, n, Q)

Q being a vector of points, this function returns the "Gram matrix" [F, G] of the cyclotomic *p*-adic height h_E with respect to the basis (ω, η) of $D = H^1_{dR}(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ given to *n p*-adic digits. In other words, if ellpadicheight(E, p, n, Q[i], Q[j]) = [f, g], corresponding to $f\omega + g\eta$ in D, then F[i, j] = f and G[i, j] = g.

```
? E = ellinit([0,0,1,-7,6]); Q = [[-2,3],[-1,3]]; p = 5; n = 5;
? [F,G] = ellpadicheightmatrix(E,p,n,Q);
? lift(F) \\ p-adic entries, integral approximation for readability
%3 =
[2364 3100]
[3100 3119]
? G
%4 =
[25225 46975]
[46975 61850]
? [F,G] * [1,-ellpadics2(E,p,n)]~
%5 =
[4 + 2*5 + 4*5^2 + 3*5^3 + 0(5^5) 4*5^2 + 4*5^3 + 5^4 + 0(5^5)]
[ 4*5^2 + 4*5^3 + 5^4 + 0(5^5) 4 + 3*5 + 4*5^2 + 4*5^3 + 5^4 + 0(5^5)]
```

ellpadiclambdamu(E, p, D, i)

Let p be a prime number and let E/\mathbb{Q} be a rational elliptic curve with good or bad multiplicative reduction at p. Return the Iwasawa invariants λ and μ for the p-adic L function $L_p(E)$, twisted by (D/.) and the *i*-th power of the Teichmüller character τ , see ellpadicL for details about $L_p(E)$.

Let χ be the cyclotomic character and choose γ in $Gal(\mathbb{Q}_p(\mu_{p^oo})/\mathbb{Q}_p)$ such that $\chi(\gamma) = 1 + 2p$. Let $L(i), D \in \mathbb{Q}_p[[X]] \otimes D_{cris}$ such that

$$(\langle \chi \rangle^{s} \tau^{i})(L^{(i),D}(\gamma-1)) = L_{p}(E, \langle \chi \rangle^{s} \tau^{i}(D/.)).$$

- When *E* has good ordinary or bad multiplicative reduction at *p*. By Weierstrass's preparation theorem the series $L^{(i),D}$ can be written $p^{\mu}(X^{\lambda} + pG(X))$ up to a *p*-adic unit, where $G(X) \in \mathbb{Z}_p[X]$. The function returns $[\lambda, \mu]$.
- When E has good supersingular reduction, we define a sequence of polynomials P_n in $\mathbb{Q}_p[X]$ of degree $< p^n$ (and bounded denominators), such that

 ${}^{L(i),D} = P_n \varphi^{n+1} \omega_E - \xi_n P_{n-1} \varphi^{n+2} \omega_E mod((1+X)^{p^n} - 1) \mathbb{Q}_p[X] \otimes D_{cris},$ where : math : ' $\xi_n = polcyclo(p^n, 1+X)$ '.Let : math : ' λ_n, μ_n 'betheinvariants of : math : ' P_n '.We find that

- μ_n is nonnegative and decreasing for *n* of given parity hence μ_{2n} tends to a limit μ^+ and μ_{2n+1} tends to a limit μ^- (both conjecturally 0).
- there exists integers λ^+ , λ^- in \mathbb{Z} (denoted with a in the reference below) such that

$$\lim_{n \to oo} \lambda_{2n} + 1/(p+1) = \lambda^+ and \lim_{n \to oo} \lambda_{2n+1} + p/(p+1) = \lambda^-.$$

The function returns : math : '[[\lambda^+, \lambda^-], [\mu^+, \mu^-]]'.

Reference: B. Perrin-Riou, Arithmétique des courbes elliptiques à réduction supersinguli\`ere en *p*, *Experimental Mathematics*, **12**, 2003, pp. 155-186.

ellpadiclog(E, p, n, P)

Given E defined over $K = \mathbb{Q}$ or \mathbb{Q}_p and P = [x, y] on E(K) in the kernel of reduction mod p, let t(P) = -x/y be the formal group parameter; this function returns L(t), where L denotes the formal logarithm (mapping the formal group of E to the additive formal group) attached to the canonical invariant differential: $dL = dx/(2y+a_1x+a_3)$.

? E = ellinit([0,0,1,-4,2]); P = [-2,1]; ? ellpadiclog(E,2,10,P) %2 = 2 + 2^3 + 2^8 + 2^9 + 2^10 + 0(2^11) ? E = ellinit([17,42]); ? p=3; Ep = ellinit(E,p); \\ E mod p ? P=[114,1218]; ellorder(Ep,P) \\ the order of P on (E mod p) is 2 %5 = 2 ? Q = ellmul(E,P,2) \\ we need a point of the form 2*P %6 = [200257/7056, 90637343/592704] ? ellpadiclog(E,3,10,Q) %7 = 3 + 2*3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 2*3^8 + 3^9 + 2*3^10 + 0(3^11)

ellpadicregulator(E, p, n, S)

Let E/\mathbb{Q} be an elliptic curve. Return the determinant of the Gram matrix of the vector of points $S = (S_1, ..., S_r)$ with respect to the "canonical" cyclotomic *p*-adic height on *E*, given to *n* (*p*-adic) digits.

When E has ordinary reduction at p, this is the expected Gram deteterminant in \mathbb{Q}_p .

In the case of supersingular reduction of E at p, the definition requires care: the regulator R is an element of $D := H_{dR}^1(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$, which is a two-dimensional \mathbb{Q}_p -vector space spanned by ω and $\eta = x\omega$ (which are defined over \mathbb{Q}) or equivalently but now over \mathbb{Q}_p by ω and $F\omega$ where F is the Frobenius endomorphism on D as defined in ellpadicfrobenius. On D we define the cyclotomic height $h_E = f\omega + g\eta$ (see ellpadicheight) and a canonical alternating bilinear form $[.,.]_D$ such that $[\omega,\eta]_D = 1$.

For any $\nu \in D$, we can define a height $h_{\nu} := [h_E, \nu]_D$ from $E(\mathbb{Q})$ to \mathbb{Q}_p and $\langle ., . \rangle_{\nu}$ the attached bilinear form. In particular, if $h_E = f\omega + g\eta$, then $h_{\eta} = [h_E, \eta]_D = f$ and $h_{\omega} = [h_E, \omega]_D = -g$ hence $h_E = h_{\eta}\omega - h_{\omega}\eta$. Then, R is the unique element of D such that

$$[\omega, \nu]_D^{r-1}[R, \nu]_D = \det(\langle S_i, S_j \rangle_{\nu})$$

for all $\nu \in D$ not in $\mathbb{Q}_p \omega$. The ellpadic regulator function returns R in the basis $(\omega, F\omega)$, which was chosen so that *p*-adic BSD conjectures are easy to state, see ellpadicbsd.

Note that by definition

$$[R,\eta]_D = \det(\langle S_i, S_j \rangle_\eta)$$

and

$$[R, \omega + \eta]_D = \det(\langle S_i, S_j \rangle_{\omega + \eta}).$$

ellpadics2(E, p, n)

If p > 2 is a prime and E/\mathbb{Q} is an elliptic curve with ordinary good reduction at p, returns the slope of the unit eigenvector of ellpadicfrobenius(E,p,n), i.e., the action of Frobenius φ on the crystalline module $D_p(E) = \mathbb{Q}_p \otimes H^1_{dR}(E/\mathbb{Q})$ in the basis of the given model $(\omega, \eta = x\omega)$, where ω is the invariant differential $dx/(2y + a_1x + a_3)$. In other words, $\eta + s_2\omega$ is an eigenvector for the unit eigenvalue of φ .

? e=ellinit([17,42]); ? ellpadics2(e,13,4) %2 = 10 + 2*13 + 6*13*3 + 0(13*4)

This slope is the unique $c \in 3^{-1}\mathbb{Z}_p$ such that the odd solution $\sigma(t) = t + O(t^2)$ of

$$-d((1)/(\sigma)(d\sigma)/(\omega)) = (x(t) + c)\omega$$

is in $t\mathbb{Z}_p[[t]]$.

It is equal to $b_2/12 - E_2/12$ where E_2 is the value of the Katz *p*-adic Eisenstein series of weight 2 on (E, ω) . This is used to construct a canonical *p*-adic height when *E* has good ordinary reduction at *p* as follows

s2 = ellpadics2(E,p,n); h(E,p,n, P, s2) = ellpadicheight(E, [p,[1,-s2]],n, P);

Since s_2 does not depend on the point P, we compute it only once.

ellperiods(w, flag, precision)

Let w describe a complex period lattice ($w = [w_1, w_2]$ or an ellinit structure). Returns normalized periods $[W_1, W_2]$ generating the same lattice such that $\tau := W_1/W_2$ has positive imaginary part and lies in the standard fundamental domain for $SL_2(\mathbb{Z})$.

If flag = 1, the function returns $[[W_1, W_2], [\eta_1, \eta_2]]$, where η_1 and η_2 are the quasi-periods attached to $[W_1, W_2]$, satisfying $\eta_2 W_1 - \eta_1 W_2 = 2i\pi$.

The output of this function is meant to be used as the first argument given to ellwp, ellzeta, ellsigma or elleisnum. Quasi-periods are needed by ellzeta and ellsigma only.

```
? L = ellperiods([1,I],1);
? [w1,w2] = L[1]; [e1,e2] = L[2];
? e2*w1 - e1*w2
%3 = 6.2831853071795864769252867665590057684*I
? ellzeta(L, 1/2 + 2*I)
%4 = 1.5707963... - 6.283185307...*I
? ellzeta([1,I], 1/2 + 2*I) \\ same but less efficient
%4 = 1.5707963... - 6.283185307...*I
```

ellpointtoz(E, P, precision)

If $E/\mathbb{C} \ \mathbb{C}/\Lambda$ is a complex elliptic curve ($\Lambda = E.omega$), computes a complex number z, well-defined modulo the lattice Λ , corresponding to the point P; i.e. such that $P = [\wp_{\Lambda}(z), \wp'_{\Lambda}(z)]$ satisfies the equation

$$y^2 = 4x^3 - g_2x - g_3,$$

where g_2 , g_3 are the elliptic invariants.

If E is defined over \mathbb{R} and $P \in E(\mathbb{R})$, we have more precisely, $0 \leq \Re(t) < w1$ and $0 \leq \Re(t) < \Im(w2)$, where (w1, w2) are the real and complex periods of E.

```
? ellpointtoz(E, [0]) \setminus the point at infinity %5 = 0
```

If E is defined over a general number field, the function returns the values corresponding to the various complex embeddings of the curve and of the point, in the same order as E.nf.roots:

```
? E=ellinit([-22032-15552*x,0], nfinit(x^2-2));
? P=[-72*x-108,0];
? ellisoncurve(E,P)
%3 = 1
? ellpointtoz(E,P)
%4 = [-0.52751724240790530394437835702346995884*I,
-0.090507650025885335533571758708283389896*I]
? E.nf.roots
%5 = [-1.4142135623730950488016887242096980786, \\ x-> -sqrt(2)
1.4142135623730950488016887242096980786] \\ x-> sqrt(2)
```

If E/\mathbb{Q}_p has multiplicative reduction, then E/\mathbb{Q}_p is analytically isomorphic to $\mathbb{Q}_p^*/q^{\mathbb{Z}}$ (Tate curve) for some *p*-adic integer *q*. The behavior is then as follows:

- If the reduction is split $(E.tate[2] \text{ is a t_PADIC})$, we have an isomorphism $\phi : E(\mathbb{Q}_p) \mathbb{Q}_p^*/q^{\mathbb{Z}}$ and the function returns $\phi(P) \in \mathbb{Q}_p$.
- If the reduction is not split (E.tate[2] is a t_POLMOD), we only have an isomorphism φ : E(K) K^{*}/q^ℤ over the unramified quadratic extension K/Q_p. In this case, the output φ(P) ∈ K is a t_POLMOD.

```
? E = ellinit([0,-1,1,0,0], O(11^5)); P = [0,0];
? [u2,u,q] = E.tate; type(u) \\ split multiplicative reduction
%2 = "t_PADIC"
? ellmul(E, P, 5) \\ P has order 5
%3 = [0]
? z = ellpointtoz(E, [0,0])
%4 = 3 + 11^2 + 2*11^3 + 3*11^4 + 6*11^5 + 10*11^6 + 8*11^7 + O(11^8)
? z^5
%5 = 1 + O(11^9)
? E = ellinit(ellfromj(1/4), O(2^6)); x=1/2; y=ellordinate(E,x)[1];
? z = ellpointtoz(E, [x,y]); \\ t_POLMOD of t_POL with t_PADIC coeffs
? liftint(z) \\ lift all p-adics
%8 = Mod(8*u + 7, u^2 + 437)
```

ellpow(*E*, *z*, *n*)

Deprecated alias for ellmul.

ellratpoints(E, h, flag)

E being an integral model of elliptic curve, return a vector containing the affine rational points on the curve of naive height less than *h*. If flag = 1, stop as soon as a point is found; return either an empty vector or a vector containing a single point. See hyperellratpoints for how *h* can be specified.

```
? E=ellinit([-25,1]);
? ellratpoints(E,10)
%2 = [[-5,1],[-5,-1],[-3,7],[-3,-7],[-1,5],[-1,-5],
[0,1],[0,-1],[5,1],[5,-1],[7,13],[7,-13]]
? ellratpoints(E,10,1)
%3 = [[-5,1]]
```

ellrootno(E, p)

E being an ell structure over \mathbb{Q} as output by ellinit, this function computes the local root number of its *L*-series at the place *p* (at the infinite place if *p* = 0). If *p* is omitted, return the global root number and in this case the curve can also be defined over a number field.

Note that the global root number is the sign of the functional equation and conjecturally is the parity of the rank of the Mordell-Weil group. The equation for E needs not be minimal at p, but if the model is already minimal the function will run faster.

ellsea(E, tors)

Let E be an *ell* structure as output by ellinit, defined over a finite field \mathbb{F}_q . This low-level function computes the order of the group $E(\mathbb{F}_q)$ using the SEA algorithm; compared to the high-level function ellcard, which includes SEA among its choice of algorithms, the tors argument allows to speed up a search for curves having almost prime order and whose quadratic twist may also have almost prime order. When tors is set to a nonzero value, the function returns 0 as soon as it detects that the order has a small prime factor not dividing tors; SEA considers modular polynomials of increasing prime degree ℓ and we return 0 as soon as we hit an ℓ (coprime to tors) dividing $\#E(\mathbb{F}_q)$:

```
? ellsea(ellinit([1,1], 2^56+3477), 1)
%1 = 72057594135613381
? forprime(p=2^128,oo, q = ellcard(ellinit([1,1],p)); if(isprime(q),break))
time = 6,571 ms.
? forprime(p=2^128,oo, q = ellsea(ellinit([1,1],p),1);if(isprime(q),break))
time = 522 ms.
```

In particular, set tors to 1 if you want a curve with prime order, to 2 if you want to allow a cofactor which is a power of two (e.g. for Edwards's curves), etc. The early exit on bad curves yields a massive speedup compared to running the cardinal algorithm to completion.

When tors is negative, similar checks are performed for the quadratic twist of the curve.

The following function returns a curve of prime order over \mathbb{F}_p .

```
cryptocurve(p) =
{
  while(1,
  my(E, N, j = Mod(random(p), p));
  E = ellinit(ellfromj(j));
  N = ellsea(E, 1); if (!N, continue);
  if (isprime(N), return(E));
  \\ try the quadratic twist for free
  if (isprime(2*p+2 - N), return(ellinit(elltwist(E))));
  );
  }
? p = randomprime([2^255, 2^256]);
? E = cryptocurve(p); \\ insist on prime order
%2 = 47,447ms
```

The same example without early abort (using ellcard(E) instead of ellsea(E, 1)) runs for about 5 minutes before finding a suitable curve.

The availability of the seadata package will speed up the computation, and is strongly recommended. The generic function ellcard should be preferred when you only want to compute the cardinal of a given curve without caring about it having almost prime order:

• If the characteristic is too small ($p \le 7$) or the field cardinality is tiny ($q \le 523$) the generic algorithm ellcard is used instead and the tors argument is ignored. (The reason for this is that SEA is not

implemented for $p \le 7$ and that if $q \le 523$ it is likely to run into an infinite loop.)

- If the field cardinality is smaller than about 2^{50} , the generic algorithm will be faster.
- Contrary to ellcard, ellsea does not store the computed cardinality in E.

ellsearch(N)

This function finds all curves in the elldata database satisfying the constraint defined by the argument N:

- if N is a character string, it selects a given curve, e.g. "11a1", or curves in the given isogeny class, e.g. "11a", or curves with given conductor, e.g. "11";
- if N is a vector of integers, it encodes the same constraints as the character string above, according to the ellconvertname correspondance, e.g. [11,0,1] for "11a1", [11,0] for "11a" and [11] for "11";
- if N is an integer, curves with conductor N are selected.

If N codes a full curve name, for instance "11a1" or [11,0,1], the output format is $[N, [a_1, a_2, a_3, a_4, a_6], G]$ where $[a_1, a_2, a_3, a_4, a_6]$ are the coefficients of the Weierstrass equation of the curve and G is a \mathbb{Z} -basis of the free part of the Mordell-Weil group attached to the curve.

? ellsearch("11a3")
%1 = ["11a3", [0, -1, 1, 0, 0], []]
? ellsearch([11,0,3])
%2 = ["11a3", [0, -1, 1, 0, 0], []]

If N is not a full curve name, then the output is a vector of all matching curves in the above format:

```
? ellsearch("11a")
%1 = [["11a1", [0, -1, 1, -10, -20], []],
["11a2", [0, -1, 1, -7820, -263580], []],
["11a3", [0, -1, 1, 0, 0], []]]
? ellsearch("11b")
%2 = []
```

ellsigma(L, z, flag, precision)

Computes the value at z of the Weierstrass σ function attached to the lattice L as given by ellperiods(, 1): including quasi-periods is useful, otherwise there are recomputed from scratch for each new z.

$$\sigma(z,L) = z \prod_{\omega \in L^*} (1 - (z)/(\omega)) e^{(z)/(\omega) + (z^2)/(2\omega^2)}.$$

It is also possible to directly input $L = [\omega_1, \omega_2]$, or an elliptic curve E as given by ellipit (L = E.omega).

```
? w = ellperiods([1,I], 1);
? ellsigma(w, 1/2)
%2 = 0.47494937998792065033250463632798296855
? E = ellinit([1,0]);
? ellsigma(E) \\ at 'x, implicitly at default seriesprecision
%4 = x + 1/60*x^5 - 1/10080*x^9 - 23/259459200*x^13 + 0(x^17)
```

If flag = 1, computes an arbitrary determination of $\log(\sigma(z))$.

ellsub(*E*, *z*1, *z*2)

Difference of the points z1 and z2 on the elliptic curve corresponding to E.

elltamagawa(E)

The object E being an elliptic curve over a number field, returns the global Tamagawa number of the curve (including the factor at infinite places).

```
? e = ellinit([1, -1, 1, -3002, 63929]); \\ curve "90c6" from elldata
? elltamagawa(e)
%2 = 288
? [elllocalred(e,p)[4] | p<-[2,3,5]]
%3 = [6, 4, 6]
? vecprod(%) \\ since e.disc > 0 the factor at infinity is 2
%4 = 144
```

elltaniyama(E, serprec)

Computes the modular parametrization of the elliptic curve E/\mathbb{Q} , where E is an ell structure as output by ellinit. This returns a two-component vector [u, v] of power series, given to n significant terms (seriesprecision by default), characterized by the following two properties. First the point (u, v) satisfies the equation of the elliptic curve. Second, let N be the conductor of E and $\Phi : X_0(N) \to E$ be a modular parametrization; the pullback by Φ of the Néron differential $du/(2v + a_1u + a_3)$ is equal to $2i\pi f(z)dz$, a holomorphic differential form. The variable used in the power series for u and v is x, which is implicitly understood to be equal to $\exp(2i\pi z)$.

The algorithm assumes that E is a *strong* Weil curve and that the Manin constant is equal to 1: in fact, $f(x) = \sum_{n>0} ellak(E, n)x^n$.

elltatepairing(E, P, Q, m)

Let E be an elliptic curve defined over a finite field k and $m \ge 1$ be an integer. This function computes the (nonreduced) Tate pairing of the points P and Q on E, where P is an m-torsion point. More precisely, let $f_{m,P}$ denote a Miller function with divisor $m[P] - m[O_E]$; the algorithm returns $f_{m,P}(Q) \in k^*/(k^*)^m$.

elltors(E)

If E is an elliptic curve defined over a number field or a finite field, outputs the torsion subgroup of E as a 3-component vector [t,v1,v2], where t is the order of the torsion group, v1 gives the structure of the torsion group as a product of cyclic groups (sorted by decreasing order), and v2 gives generators for these cyclic groups. E must be an ell structure as output by ellinit.

? E = ellinit([-1,0]); ? elltors(E) %1 = [4, [2, 2], [[0, 0], [1, 0]]]

Here, the torsion subgroup is isomorphic to $\mathbb{Z}/2\mathbb{Z}x\mathbb{Z}/2\mathbb{Z}$, with generators [0,0] and [1,0].

elltwist(E, P)

Returns the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of the twist of the elliptic curve E by the quadratic extension of the coefficient ring defined by P (when P is a polynomial) or quadpoly(P) when P is an integer. If E is defined over a finite field, then P can be omitted, in which case a random model of the unique nontrivial twist is returned. If E is defined over a number field, the model should be replaced by a minimal model (if one exists).

Example: Twist by discriminant -3:

? elltwist(ellinit([0,a2,0,a4,a6]),-3) %1 = [0,-3*a2,0,9*a4,-27*a6]

Twist by the Artin-Schreier extension given by $x^2 + x + T$ in characteristic 2:

```
? lift(elltwist(ellinit([a1,a2,a3,a4,a6]*Mod(1,2)),x^2+x+T))
%1 = [a1,a2+a1^2*T,a3,a4,a6+a3^2*T]
```

Twist of an elliptic curve defined over a finite field:

```
? E=ellinit([1,7]*Mod(1,19));lift(elltwist(E))
%1 = [0,0,0,11,12]
```

ellweilcurve(E, ms)

If E' is an elliptic curve over \mathbb{Q} , let $L_{E'}$ be the sub- \mathbb{Z} -module of $\operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, \mathbb{Q})$ attached to E' (It is given by x[3] if [M, x] = msfromell(E').)

On the other hand, if N is the conductor of E and f is the modular form for $\Gamma_0(N)$ attached to E, let L_f be the lattice of the f-component of $\operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, \mathbb{Q})$ given by the elements ϕ such that $\phi(0, \gamma^{-1}0) \in \mathbb{Z}$ for all $\gamma \in \Gamma_0(N)$ (see mslattice).

Let E' run through the isomorphism classes of elliptic curves isogenous to E as given by ellisomat (and in the same order). This function returns a pair [vE, vS] where vE contains minimal models for the E' and vS contains the list of Smith invariants for the lattices $L_{E'}$ in L_f . The function also accepts the output of ellisomat, i.e. the isogeny class. If the optional argument ms is present, it contains the output of msfromell(vE, 0), i.e. the new modular symbol space M of level N and a vector of triples $[x^+, x^-, L]$ attached to each curve E'.

In particular, the strong Weil curve amongst the curves isogenous to E is the one whose Smith invariants are [c, c], where c is the Manin constant, conjecturally equal to 1.

? E = ellinit("11a3"); ? [vE, vS] = ellweilcurve(E); ? [n] = [i | i<-[1..#vS], vS[i]==[1,1]] \\ lattice with invariant [1,1] %3 = [2]? ellidentify(vE[n]) \\ ... corresponds to strong Weil curve %4 = [["11a1", [0, -1, 1, -10, -20], []], [1, 0, 0, 0]]? [vE, vS] = ellweilcurve(E, &ms); \\ vE,vS are as above ? [M, vx] = ms; msdim(M) \\ ... but ms contains more information %6 = 3 ? #vx %7 = 3 ? vx[1] %8 = [[1/25, -1/10, -1/10]~, [0, 1/2, -1/2]~, [1/25,0; -3/5,1; 2/5,-1]] ? forell(E, 11,11, print(msfromell(ellinit(E[1]), 1)[2])) $[1/5, -1/2, -1/2] \sim$ [1, -5/2, -5/2]~ $[1/25, -1/10, -1/10] \sim$

The last example prints the modular symbols x^+ in M^+ attached to the curves 11a1, 11a2 and 11a3.

ellweilpairing(E, P, Q, m)

Let E be an elliptic curve defined over a finite field and $m \ge 1$ be an integer. This function computes the Weil pairing of the two m-torsion points P and Q on E, which is an alternating bilinear map. More precisely, let $f_{m,R}$ denote a Miller function with divisor $m[R] - m[O_E]$; the algorithm returns the m-th root of unity

$$\varepsilon(P,Q)^m \cdot f_{m,P}(Q) / f_{m,Q}(P),$$

where f(R) is the extended evaluation of f at the divisor $[R] - [O_E]$ and $\varepsilon(P, Q) \in 1$ is given by Weil reciprocity: $\varepsilon(P, Q) = 1$ if and only if P, Q, O_E are not pairwise distinct.

ellwp(*w*, *z*, *flag*, *precision*)

Computes the value at z of the Weierstrass \wp function attached to the lattice w as given by ellperiods. It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by elliptic (w = E.omega).

```
? w = ellperiods([1,I]);
? ellwp(w, 1/2)
%2 = 6.8751858180203728274900957798105571978
? E = ellinit([1,1]);
? ellwp(E, 1/2)
%4 = 3.9413112427016474646048282462709151389
```

One can also compute the series expansion around z = 0:

```
? E = ellinit([1,0]);

? ellwp(E) \\ 'x implicitly at default seriesprecision

%5 = x^{-2} - 1/5*x^{2} + 1/75*x^{6} - 2/4875*x^{10} + 0(x^{14})

? ellwp(E, x + 0(x^{12})) \\ explicit precision

%6 = x^{-2} - 1/5*x^{2} + 1/75*x^{6} + 0(x^{9})
```

Optional *flag* means 0 (default): compute only $\wp(z)$, 1: compute $[\wp(z), \wp'(z)]$.

For instance, the Dickson elliptic functions sm and sn can be implemented as follows

```
smcm(z) =
{ my(a, b, E = ellinit([0,-1/(4*27)])); \\ ell. invariants (g2,g3)=(0,1/27)
[a,b] = ellwp(E, z, 1);
[6*a / (1-3*b), (3*b+1)/(3*b-1)];
}
? [s,c] = smcm(0.5);
? s
%2 = 0.4898258757782682170733218609
? c
%3 = 0.9591820206453842491187464098
? s^3+c^3
%4 = 1.0000000000000000000000
? smcm('x + 0('x^11))
%5 = [x - 1/6*x^4 + 2/63*x^7 - 13/2268*x^10 + 0(x^11),
1 - 1/3*x^3 + 1/18*x^6 - 23/2268*x^9 + 0(x^10)]
```

ellxn(E, n, v)

For any affine point P = (t, u) on the curve E, we have

$$[n]P = (\phi_n(P)\psi_n(P):\omega_n(P):\psi_n(P)^3)$$

for some ϕ_n, ω_n, ψ_n in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][t, u]$ modulo the curve equation. This function returns a pair [A, B] of polynomials in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][v]$ such that $[A(t), B(t)] = [\phi_n(P), \psi_n(P)^2]$ in the function field of E, whose quotient give the abscissa of [n]P. If P is an n-torsion point, then B(t) = 0.

```
? E = ellinit([17,42]); [t,u] = [114,1218];
? T = ellxn(E, 2, 'X)
%2 = [X^4 - 34*X^2 - 336*X + 289, 4*X^3 + 68*X + 168]
? [a,b] = subst(T,'X,t);
%3 = [168416137, 5934096]
? a / b == ellmul(E, [t,u], 2)[1]
%4 = 1
```

ellzeta(w, z, precision)

Computes the value at z of the Weierstrass ζ function attached to the lattice w as given by ellperiods(, 1):

including quasi-periods is useful, otherwise there are recomputed from scratch for each new z.

$$\zeta(z,L) = (1)/(z) + z^2 \sum_{\omega \in L^*} (1)/(\omega^2(z-\omega)).$$

It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by elliptic (w = E.omega). The quasi-periods of ζ , such that

$$\zeta(z + a\omega_1 + b\omega_2) = \zeta(z) + a\eta_1 + b\eta_2$$

for integers a and b are obtained as $\eta_i = 2\zeta(\omega_i/2)$. Or using directly elleta.

```
? w = ellperiods([1,I],1);
? ellzeta(w, 1/2)
%2 = 1.5707963267948966192313216916397514421
? E = ellinit([1,0]);
? ellzeta(E, E.omega[1]/2)
%4 = 0.84721308479397908660649912348219163647
```

One can also compute the series expansion around z = 0 (the quasi-periods are useless in this case):

```
? E = ellinit([0,1]);
? ellzeta(E) \\ at 'x, implicitly at default seriesprecision
%4 = x^-1 + 1/35*x^5 - 1/7007*x^11 + 0(x^15)
? ellzeta(E, x + 0(x^20)) \\ explicit precision
%5 = x^-1 + 1/35*x^5 - 1/7007*x^11 + 1/1440257*x^17 + 0(x^18)
```

ellztopoint(E, z, precision)

E being an *ell* as output by ellinit, computes the coordinates [x, y] on the curve *E* corresponding to the complex or *p*-adic parameter *z*. Hence this is the inverse function of ellpointtoz.

• If E is defined over a p-adic field and has multiplicative reduction, then z is understood as an element on the Tate curve $\bar{Q}_n^*/q^{\mathbb{Z}}$.

```
? E = ellinit([0,-1,1,0,0], 0(11^5));
? [u2,u,q] = E.tate; type(u)
%2 = "t_PADIC" \\ split multiplicative reduction
? z = ellpointtoz(E, [0,0])
%3 = 3 + 11^2 + 2*11^3 + 3*11^4 + 6*11^5 + 10*11^6 + 8*11^7 + 0(11^8)
? ellztopoint(E,z)
%4 = [0(11^9), 0(11^9)]
? E = ellinit(ellfromj(1/4), 0(2^6)); x=1/2; y=ellordinate(E,x)[1];
? z = ellpointtoz(E,[x,y]); \\ nonsplit: t_POLMOD with t_PADIC coefficients
? P = ellztopoint(E, z);
? P[1] \\ y coordinate is analogous, more complicated
%8 = Mod(0(2^4)*x + (2^-1 + 0(2^5)), x^2 + (1 + 2^2 + 2^4 + 2^5 + 0(2^7)))
```

• If E is defined over the complex numbers (for instance over \mathbb{Q}), z is understood as a complex number in \mathbb{C}/Λ_E . If the short Weierstrass equation is $y^2 = 4x^3 - g_2x - g_3$, then [x, y] represents the Weierstrass \wp -function and its derivative. For a general Weierstrass equation we have

 $x = \wp(z) - b_2/12, y = \wp'(z)/2 - (a_1x + a_3)/2.$ If : math : 'z'isinthelattice defining : math : 'E'over : math : 'C', the result is the point at infinity : math : '[0]'.

erfc(x, precision)

Complementary error function, analytic continuation of $(2/\sqrt{\pi}) \int_x^o oe^{-t^2} dt = sign(x)incgam(1/2, x^2)/\sqrt{\pi}$ for real x! = 0. The latter expression extends the function definition from real x to complex x with positive real part (or zero real part and positive imaginary part). This is extended to the whole complex plane by the functional equation erfc(-x) = 2 - erfc(x).

errname(E)

Returns the type of the error message E as a string.

```
? iferr(1 / 0, E, print(errname(E)))
e_INV
? ?? e_INV
[...]
* "e_INV". Tried to invert a noninvertible object x in function s.
[...]
```

eta(z, flag, precision)

Variants of Dedekind's η function. If flag = 0, return $\prod_{n=1}^{o} o(1 - q^n)$, where q depends on x in the following way:

- $q = e^{2i\pi x}$ if x is a *complex number* (which must then have positive imaginary part); notice that the factor $q^{1/24}$ is missing!
- q = x if x is a t_PADIC, or can be converted to a *power series* (which must then have positive valuation).

If *flag* is nonzero, x is converted to a complex number and we return the true η function, $q^{1/24} \prod_{n=1}^{o} o(1-q^n)$, where $q = e^{2i\pi x}$.

eulerfrac(n)

Euler number E_n , where $E_0 = 1$, $E_1 = 0$, $E_2 = -1$,..., are integers such that

$$(1)/(\cosh t) = \sum_{n>=0} (E_n)/(n!)t^n.$$

The argument n should be a nonnegative integer.

```
? vector(10,i,eulerfrac(i))
%1 = [0, -1, 0, 5, 0, -61, 0, 1385, 0, -50521]
? eulerfrac(20000);
? sizedigit(%))
%3 = 73416
```

eulerianpol(n, v)

Eulerian polynomial A_n in variable v.

? eulerianpol(2) %1 = x + 1 ? eulerianpol(5, 't) %2 = t^4 + 26*t^3 + 66*t^2 + 26*t + 1

eulerphi(x)

Euler's ϕ (totient) function of the integer ||x||, in other words $||(\mathbb{Z}/x\mathbb{Z})^*||$.

? eulerphi(40) %1 = 16

According to this definition we let $\phi(0) := 2$, since $\mathbb{Z}^* = -1, 1$; this is consistent with znstar(0): we have znstar:math:`(n).no = eulerphi(n)` for all $n \in \mathbb{Z}$.

eulerpol(n, v)

Euler polynomial E_n in variable v.

? eulerpol(1)
%1 = x - 1/2
? eulerpol(3)
%2 = x^3 - 3/2*x^2 + 1/4

eulervec(n)

Returns a vector containing, as rational numbers, the nonzero Euler numbers E_0, E_2, \ldots, E_{2n} :

```
? eulervec(5) \\ E_0, E_2..., E_10
%1 = [1, -1, 5, -61, 1385, -50521]
? eulerfrac(10)
%2 = -50521
```

This routine uses more memory but is a little faster than repeated calls to eulerfrac:

```
? forstep(n = 2, 8000, 2, eulerfrac(n))
time = 46,851 ms.
? eulervec(4000);
time = 30,588 ms.
```

exp(x, precision)

Exponential of x. p-adic arguments with positive valuation are accepted.

expm1(*x*, *precision*)

Return $\exp(x) - 1$, computed in a way that is also accurate when the real part of x is near 0. A naive direct computation would suffer from catastrophic cancellation; PARI's direct computation of $\exp(x)$ alleviates this well known problem at the expense of computing $\exp(x)$ to a higher accuracy when x is small. Using expm1 is recommended instead:

```
? default(realprecision, 10000); x = 1e-100;
? a = expm1(x);
time = 4 ms.
? b = exp(x)-1;
time = 4 ms.
? default(realprecision, 10040); x = 1e-100;
? c = expm1(x); \\ reference point
? abs(a-c)/c \\ relative error in expm1(x)
%7 = 1.4027986153764843997 E-10019
? abs(b-c)/c \\ relative error in exp(x)-1
%8 = 1.7907031188259675794 E-9919
```

As the example above shows, when x is near 0, expm1 is more accurate than exp(x)-1.

exponent(x)

When x is a t_REAL, the result is the binary exponent e of x. For a nonzero x, this is the unique integer e such that $2^e \le ||x|| \le 2^{e+1}$. For a real 0, this returns the PARI exponent e attached to x (which may represent any floating-point number less than 2^e in absolute value).

? exponent(Pi) %1 = 1 ? exponent(4.0) %2 = 2 ? exponent(0.0) %3 = -128 ? default(realbitprecision) %4 = 128

This definition extends naturally to nonzero integers, and the exponent of an exact 0 is -oo by convention.

For convenience, we *define* the exponent of a t_FRAC a/b as the difference of exponent(a) and exponent(b); note that, if e' denotes the exponent of :math: `a/b * 1.0`, then the exponent e we return is either e' or e' + 1, thus 2^{e+1} is an upper bound for ||a/b||.

```
? [ exponent(9), exponent(10), exponent(9/10), exponent(9/10*1.) ]
%5 = [3, 3, 0, -1]
```

For a PARI object of type t_COMPLEX, t_POL, t_SER, t_VEC, t_COL, t_MAT this returns the largest exponent found among the components of x. Hence 2^{e+1} is a quick upper bound for the sup norm of real matrices or polynomials; and $2^{e+(3/2)}$ for complex ones.

```
? exponent(3*x^2 + 15*x - 100)
%5 = 6
? exponent(0)
%6 = -00
```

exportall()

Declare all current dynamic variables as exported variables. Such variables are visible inside parallel sections in place of global variables.

? fun(x)=x^2+1; ? parvector(10,i,fun(i)) *** mt: please use export(fun). ? exportall()

<pre>? parvector(10,i,fun(i))</pre>
%4 = [2,5,10,17,26,37,50,65,82,101]

extern(str)

The string *str* is the name of an external command (i.e. one you would type from your UNIX shell prompt). This command is immediately run and its output fed into **gp**, just as if read from a file.

externstr(str)

The string *str* is the name of an external command (i.e. one you would type from your UNIX shell prompt). This command is immediately run and its output is returned as a vector of GP strings, one component per output line.

factor(x, D)

Factor x over domain D; if D is omitted, it is determined from x. For instance, if x is an integer, it is factored in \mathbb{Z} , if it is a polynomial with rational coefficients, it is factored in $\mathbb{Q}[x]$, etc., see below for details. The result is a two-column matrix: the first contains the irreducibles dividing x (rational or Gaussian primes, irreducible polynomials), and the second the exponents. By convention, 0 is factored as 0^1 .

:math:`x in mathbb{Q}.` See factorint for the algorithms used. The factorization includes the unit -1 when x < 0 and all other factors are positive; a denominator is factored with negative exponents. The factors are sorted in increasing order.

? factor(-7/106)
%1 =
[-1 1]
[2 -1]
[7 1]
[53 -1]

By convention, 1 is factored as matrix(0,2) (the empty factorization, printed as [;]).

Large rational "primes" > 2^{64} in the factorization are in fact *pseudoprimes* (see ispseudoprime), a priori not rigorously proven primes. Use isprime to prove primality of these factors, as in

```
? fa = factor(2^2^7 + 1)
%2 =
[59649589127497217 1]
[5704689200685129054721 1]
? isprime( fa[,1] )
%3 = [1, 1]~ \\ both entries are proven primes
```

Another possibility is to globally set the default factor_proven, which will perform a rigorous primality proof for each pseudoprime factor but will slow down PARI.

A t_INT argument D can be added, meaning that we only trial divide by all primes p < D and the addprimes entries, then skip all expensive factorization methods. The limit D must be nonnegative. In this case, one entry in the factorization may be a composite number: all factors less than D^2 and primes from the addprimes table are actual primes. But (at most) one entry may not verify this criterion, and it may be prime or composite: it is only known to be coprime to all other entries and not a pure power.

```
? factor(2^2^7 +1, 10^5)
%4 =
[340282366920938463463374607431768211457 1]
```

Deprecated feature. Setting D = 0 is the same as setting it to *primelimit* + 1.

This routine uses trial division and perfect power tests, and should not be used for huge values of D (at most 10^9 , say): factorint(, 1 + 8) will in general be faster. The latter does not guarantee that all small prime factors are found, but it also finds larger factors and in a more efficient way.

```
? F = (2^2^7 + 1) * 1009 * (10^5+3); factor(F, 10^5) \\ fast, incomplete
time = 0 \text{ ms}.
%5 =
[1009 1]
[34029257539194609161727850866999116450334371 1]
? factor(F, 10^9) \\ slow
time = 3,260 ms.
%6 =
[1009 1]
[100003 1]
[340282366920938463463374607431768211457 1]
? factorint(F, 1+8) \setminus much faster and all small primes were found
time = 8 \text{ ms}.
%7 =
[1009 1]
[100003 1]
[340282366920938463463374607431768211457 1]
? factor(F) \\ complete factorization
time = 60 \text{ ms}.
%8 =
[1009 1]
[100003 1]
[59649589127497217 1]
[5704689200685129054721 1]
```

Setting D = I will factor in the Gaussian integers $\mathbb{Z}[i]$:

:math:`x in mathbb{Q} (i).` The factorization is performed with Gaussian primes in $\mathbb{Z}[i]$ and includes Gaussian units in 1, *i*; factors are sorted by increasing norm. Except for a possible leading unit, the Gaussian factors are normalized: rational factors are positive and irrational factors have positive imaginary part (a canonical representa.

Unless factor_proven is set, large factors are actually pseudoprimes, not proven primes; a rational factor is prime if less than 2^{64} and an irrational one if its norm is less than 2^{64} .

? factor(5*I) %9 = [2 + I 1] [1 + 2*I 1]

One can force the factorization of a rational number by setting the domain D = I:

? factor(-5, I)
%10 =
[I 1]
[2 + I 1]
[1 + 2*I 1]
? factorback(%)
%11 = -5

Univariate polynomials and rational functions. PARI can factor univariate polynomials in K[t]. The following base fields K are currently supported: $\mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{Q}_p$, finite fields and number fields. See factormod and factorff for the algorithms used over finite fields and nffactor for the algorithms over number fields. The irreducible factors are sorted by increasing degree and normalized: they are monic except when $K = \mathbb{Q}$ where they are primitive in $\mathbb{Z}[t]$.

The content is *not* included in the factorization, in particular factorback will in general recover the original x only up to multiplication by an element of K^* : when $K! = \mathbb{Q}$, this scalar is pollead(x) (since irreducible factors are monic); and when $K = \mathbb{Q}$ you can either ask for the \mathbb{Q} -content explicitly of use factorback:

```
? P = t^2 + 5*t/2 + 1; F = factor(P)
%12 =
[t + 2 1]
[2*t + 1 1]
? content(P, 1) \\ Q-content
%13 = 1/2
? pollead(factorback(F)) / pollead(P)
%14 = 2
```

You can specify K using the optional "domain" argument D as follows

- $K = \mathbb{Q}$: D a rational number (t_INT or t_FRAC),
- $K = \mathbb{Z}/p\mathbb{Z}$ with p prime : D a t_INTMOD modulo p; factoring modulo a composite number is not supported.
- $K = \mathbb{F}_q$: D a t_FFELT encoding the finite field; you can also use a t_POLMOD of t_INTMOD modulo a prime p but this is usually less convenient;
- $K = \mathbb{Q}[X]/(T)$ a number field : D a t_POLMOD modulo T,
- $K = \mathbb{Q}(i)$ (alternate syntax for special case): D = I,
- $K = \mathbb{Q}(w)$ a quadratic number field (alternate syntax for special case): D a t_QUAD,
- $K = \mathbb{R} : D$ a real number (t_REAL); truncate the factorization at accuracy $\operatorname{precision}(D)$. If x is inexact and $\operatorname{precision}(x)$ is less than $\operatorname{precision}(D)$, then the precision of x is used instead.

- $K = \mathbb{C}$: D a complex number with a t_REAL component, e.g. I * 1.; truncate the factorization as for $K = \mathbb{R}$,
- K = Q_p : D a t_PADIC; truncate the factorization at p-adic accuracy padicprec(D), possibly less if x is inexact with insufficient p-adic accuracy;

```
? T = x^2+1;
? factor(T, 1); \\ over Q
? factor(T, Mod(1,3)) \\ over F_3
? factor(T, ffgen(ffinit(3,2,'t))^0) \\ over F_{3^2}
? factor(T, Mod(Mod(1,3), t^2+t+2)) \\ over F_{3^2}, again
? factor(T, 0(3^6)) \\ over Q_3, precision 6
? factor(T, 1.) \\ over R, current precision
? factor(T, I*1.) \\ over C
? factor(T, Mod(1, y^3-2)) \\ over Q(2^{1/3})
```

In most cases, it is possible and simpler to call a specialized variant rather than use the above scheme:

```
? factormod(T, 3) \\ over F_3
? factormod(T, [t^2+t+2, 3]) \\ over F_{3^2}
? factormod(T, ffgen(3^2, 't)) \\ over F_{3^2}
? factorpadic(T, 3,6) \\ over Q_3, precision 6
? nffactor(y^3-2, T) \\ over Q(2^{1/3})
? polroots(T) \\ over C
? polrootsreal(T) \\ over R (real polynomial)
```

It is also possible to let the routine use the smallest field containing all coefficients, taking into account quotient structures induced by t_INTMOD s and t_POLMOD s (e.g. if a coefficient in $\mathbb{Z}/n\mathbb{Z}$ is known, all rational numbers encountered are first mapped to $\mathbb{Z}/n\mathbb{Z}$; different moduli will produce an error):

```
? T = x^2+1;
? factor(T); \\ over Q
? factor(T*Mod(1,3)) \\ over F_3
? factor(T*ffgen(ffinit(3,2,'t))^0) \\ over F_{3^2}
? factor(T*Mod(Mod(1,3), t^2+t+2)) \\ over F_{3^2}, again
? factor(T*(1 + 0(3^6)) \\ over Q_3, precision 6
? factor(T*1.) \\ over R, current precision
? factor(T*(1.+0.*I)) \\ over C
? factor(T*Mod(1, y^3-2)) \\ over Q(2^{1/3})
```

Multiplying by a suitable field element equal to $1 \in K$ in this way is error-prone and is not recommanded. Factoring existing polynomials with obvious fields of coefficients is fine, the domain argument D should be used instead ad hoc conversions.

Note on inexact polynomials. Polynomials with inexact coefficients (e.g. floating point or *p*-adic numbers) are first rounded to an exact representation, then factored to (potentially) infinite accuracy and we return a truncated approximation of that virtual factorization. To avoid pitfalls, we advise to only factor *exact* polynomials:

? factor(x^2-1+0(2^2)) \\ rounded to x^2 + 3, irreducible in Q_2
%1 =
[(1 + 0(2^2))*x^2 + 0(2^2)*x + (1 + 2 + 0(2^2)) 1]
? factor(x^2-1+0(2^3)) \\ rounded to x^2 + 7, reducible !
%2 =
[(1 + 0(2^3))*x + (1 + 2 + 0(2^3)) 1]

 $[(1 + 0(2^3))*x + (1 + 2^2 + 0(2^3)) 1]$? factor(x^2-1, 0(2^2)) \\ no ambiguity now
%3 =
[(1 + 0(2^2))*x + (1 + 0(2^2)) 1]
[(1 + 0(2^2))*x + (1 + 2 + 0(2^2)) 1]

Note about inseparable polynomials. Polynomials with inexact coefficients are considered to be squarefree: indeed, there exist a squarefree polynomial arbitrarily close to the input, and they cannot be distinguished at the input accuracy. This means that irreducible factors are repeated according to their apparent multiplicity. On the contrary, using a specialized function such as factorpadic with an *exact* rational input yields the correct multiplicity when the (now exact) input is not separable. Compare:

? factor(z² + 0(5²)))
%1 =
[(1 + 0(5²))*z + 0(5²) 1]
[(1 + 0(5²))*z + 0(5²) 1]
? factor(z², 0(5²))
%2 =
[1 + 0(5²))*z + 0(5²) 2]

Multivariate polynomials and rational functions. PARI recursively factors *multivariate* polynomials in $K[t_1, ..., t_d]$ for the same fields K as above and the argument D is used in the same way to specify K. The irreducible factors are sorted by their main variable (least priority first) then by increasing degree.

```
? factor(x^2 + y^2, Mod(1,5))
%1 =
[ x + Mod(2, 5)*y 1]
[Mod(1, 5)*x + Mod(3, 5)*y 1]
? factor(x^2 + y^2, 0(5^2))
%2 =
[ (1 + 0(5^2))*x + (0(5^2)*y^2 + (2 + 5 + 0(5^2))*y + 0(5^2)) 1]
[(1 + 0(5^2))*x + (0(5^2)*y^2 + (3 + 3*5 + 0(5^2))*y + 0(5^2)) 1]
? lift(%)
%3 =
[ x + 7*y 1]
[x + 18*y 1]
```

Note that the implementation does not really support inexact real fields (\mathbb{R} or \mathbb{C}) and usually misses factors even if the input is exact:

? factor(x² + y², I) \\ over Q(i) %4 = [x - I*y 1]

[x + I*y 1]
? factor(x^2 + y^2, I*1.) \\ over C
%5 =
[x^2 + y^2 1]

factorback(f, e)

Gives back the factored object corresponding to a factorization. The integer 1 corresponds to the empty factorization.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization, as produced with any factor command. A few examples:

```
? factor(12)
%1 =
[2 2]
[3 1]
? factorback(%)
%2 = 12
? factorback([2,3], [2,1]) \\ 2^3 * 3^1
%3 = 12
? factorback([5,2,3])
%4 = 30
```

factorcantor(x, p)

This function is obsolete, use factormod.

factorff(x, p, a)

Obsolete, kept for backward compatibility: use factormod.

factorial(x, precision)

Factorial of x. The expression x! gives a result which is an integer, while factorial(x) gives a real number.

factorint(x, flag)

Factors the integer n into a product of pseudoprimes (see ispseudoprime), using a combination of the Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers. The output is a two-column matrix as for factor: the first column contains the "prime" divisors of n, the second one contains the (positive) exponents.

By convention 0 is factored as 0^1 , and 1 as the empty factorization; also the divisors are by default not proven primes if they are larger than 2^{64} , they only failed the BPSW compositeness test (see ispseudoprime). Use isprime on the result if you want to guarantee primality or set the factor_proven default to 1. Entries of the private prime tables (see addprimes) are also included as is.

This gives direct access to the integer factoring engine called by most arithmetical functions. *flag* is optional; its binary digits mean 1: avoid MPQS, 2: skip first stage ECM (we may still fall back to it later), 4: avoid Rho and SQUFOF, 8: don't run final ECM (as a result, a huge composite may be declared to be prime). Note that a (strong) probabilistic primality test is used; thus composites might not be detected, although no example is known.

You are invited to play with the flag settings and watch the internals at work by using gp's debug default parameter (level 3 shows just the outline, 4 turns on time keeping, 5 and above show an increasing amount of internal details).

factormod(f, D, flag)

Factors the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T(y) an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[y]/(T)$ (as usual the main variable of T must have lower priority than the main variable of f);
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

The coefficients of f must be operation-compatible with the corresponding finite field. The result is a twocolumn matrix, the first column being the irreducible polynomials dividing f, and the second the exponents. By convention, the 0 polynomial factors as 0^1 ; a nonzero constant polynomial has empty factorization, a $0x^2$ matrix. The irreducible factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? factormod(x^2 + 1, 3) \\ over F_3
%1 =
[Mod(1, 3)*x^2 + Mod(1, 3) 1]
? liftall( factormod(x^2 + 1, [t^2+1, 3]) ) \\ over F_9
%2 =
[ x + t 1]
[x + 2*t 1]
\\ same, now letting GP choose a model
? T = ffinit(3,2,'t)
\%3 = Mod(1, 3)*t^2 + Mod(1, 3)*t + Mod(2, 3)
? liftall( factormod(x^2 + 1, [T, 3]) )
\%4 = \setminus t is a root of T !
[x + (t + 2) 1]
[x + (2*t + 1) 1]
? t = ffgen(t^2+Mod(1,3)); factormod(x^2 + t^0) \\ same using t_FFELT
%5 =
[x + t 1]
[x + 2*t 1]
? factormod(x^2+Mod(1,3))
%6 =
[Mod(1, 3)*x^2 + Mod(1, 3) 1]
? liftall( factormod(x^2 + Mod(Mod(1,3), y^2+1)) )
%7 =
[x + y 1]
[x + 2*y 1]
```

If flag is nonzero, outputs only the *degrees* of the irreducible polynomials (for example to compute an *L*-function). By convention, a constant polynomial (including the 0 polynomial) has empty factorization. The degrees appear in increasing order but need not correspond to the ordering with flag = 0 when multiplicities are present.

```
? f = x^3 + 2*x^2 + x + 2;
? factormod(f, 5) \\ (x+2)^2 * (x+3)
%1 =
[Mod(1, 5)*x + Mod(2, 5) 2]
[Mod(1, 5)*x + Mod(3, 5) 1]
? factormod(f, 5, 1) \\ (deg 1) * (deg 1)^2
%2 =
[1 1]
[1 2]
```

factormodDDF(f, D)

Distinct-degree factorization of the squarefree polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[x]/(T)$;
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

This is somewhat faster than full factorization. The coefficients of f must be operation-compatible with the corresponding finite field. The result is a two-column matrix:

- the first column contains monic (squarefree) pairwise coprime polynomials dividing f, all of whose irreducible factors have degree d;
- the second column contains the degrees of the irreducible factors.

The factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? f = (x^2 + 1) * (x^2-1);
? factormodSQF(f,3) \setminus squarefree over F_3
%2 =
[Mod(1, 3)*x^4 + Mod(2, 3) 1]
? factormodDDF(f, 3)
%3 =
[Mod(1, 3)*x^2 + Mod(2, 3) 1] \setminus two degree 1 factors
[Mod(1, 3)*x^2 + Mod(1, 3) 2] \setminus irred of degree 2
? for(i=1,10^5,factormodDDF(f,3))
time = 424 \text{ ms}.
? for(i=1,10^5,factormod(f,3)) \\ full factorization is slower
time = 464 \text{ ms}.
? liftall( factormodDDF(x^2 + 1, [3, t^2+1]) ) \\ over F_9
%6 =
[x^2 + 1 \ 1] \setminus  product of two degree 1 factors
? t = ffgen(t^2+Mod(1,3)); factormodDDF(x^2 + t^0) \\ same using t_FFELT
%7 =
```

[x^2 + 1 1]
? factormodDDF(x^2-Mod(1,3))
%8 =
[Mod(1, 3)*x^2 + Mod(2, 3) 1]

factormodSQF(f, D)

Squarefree factorization of the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[x]/(T)$;
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

This is somewhat faster than full factorization. The coefficients of f must be operation-compatible with the corresponding finite field. The result is a two-column matrix:

- the first column contains monic squarefree pairwise coprime polynomials dividing f;
- the second column contains the power to which the polynomial in column 1 divides f;

The factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? f = (x^2 + 1)^3 * (x^2-1)^2;
? factormodSQF(f, 3) \\ over F_3
%1 =
[Mod(1, 3)*x^2 + Mod(2, 3) 2]
[Mod(1, 3)*x^2 + Mod(1, 3) 3]
? for(i=1,10^5,factormodSQF(f,3))
time = 192 \text{ ms.}
? for(i=1,10^5,factormod(f,3)) \\ full factorization is slower
time = 409 \text{ ms}.
? liftall( factormodSQF((x<sup>2</sup> + 1)<sup>3</sup>, [3, t<sup>2</sup>+1]) ) \\ over F_9
%4 =
[x^2 + 1 3]
? t = ffgen(t<sup>2</sup>+Mod(1,3)); factormodSQF((x^2 + t^0)<sup>3</sup>) \\ same using t_FFELT
%5 =
[x^2 + 1 3]
? factormodSQF(x^8 + x^7 + x^6 + x^2 + x + Mod(1,2))
%6 =
[Mod(1, 2)*x + Mod(1, 2) 2]
[Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2) 3]
```

factornf(x, t)

This function is obsolete, use nffactor.

factorization of the univariate polynomial x over the number field defined by the (univariate) polynomial t. x may have coefficients in \mathbb{Q} or in the number field. The algorithm reduces to factorization over \mathbb{Q} (Trager's trick). The direct approach of nffactor, which uses van Hoeij's method in a relative setting, is in general faster.

The main variable of t must be of *lower* priority than that of x (see priority (in the PARI manual)). However if nonrational number field elements occur (as polmods or polynomials) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t. For example

factorpadic(pol, p, r)

p-adic factorization of the polynomial *pol* to precision *r*, the result being a two-column matrix as in factor. Note that this is not the same as a factorization over $\mathbb{Z}/p^r\mathbb{Z}$ (polynomials over that ring do not form a unique factorization domain, anyway), but approximations in $\mathbb{Q}/p^r\mathbb{Z}$ of the true factorization in $\mathbb{Q}_p[X]$.

```
? factorpadic(x^2 + 9, 3,5)
%1 =
[(1 + 0(3^5))*x^2 + 0(3^5)*x + (3^2 + 0(3^5)) 1]
? factorpadic(x^2 + 1, 5,3)
%2 =
[ (1 + 0(5^3))*x + (2 + 5 + 2*5^2 + 0(5^3)) 1]
[(1 + 0(5^3))*x + (3 + 3*5 + 2*5^2 + 0(5^3)) 1]
```

The factors are normalized so that their leading coefficient is a power of p. The method used is a modified version of the round 4 algorithm of Zassenhaus.

If *pol* has inexact t_PADIC coefficients, this is not always well-defined; in this case, the polynomial is first made integral by dividing out the *p*-adic content, then lifted to \mathbb{Z} using truncate coefficientwise. Hence we actually factor exactly a polynomial which is only *p*-adically close to the input. To avoid pitfalls, we advise to only factor polynomials with exact rational coefficients.

ffcompomap(f, g)

Let k, l, m be three finite fields and f a (partial) map from l to m and g a (partial) map from k to l, return the (partial) map f og from k to m.

```
a = ffgen([3,5],'a); b = ffgen([3,10],'b); c = ffgen([3,20],'c);
m = ffembed(a, b); n = ffembed(b, c);
rm = ffinvmap(m); rn = ffinvmap(n);
nm = ffcompomap(n,m);
ffmap(n,ffmap(m,a)) == ffmap(nm, a)
%5 = 1
ffcompomap(rm, rn) == ffinvmap(nm)
%6 = 1
```

ffembed(*a*, *b*)

Given two finite fields elements a and b, return a *map* embedding the definition field of a to the definition field of b. Assume that the latter contains the former.

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? A = ffmap(m, a); ? minpoly(A) == minpoly(a) %5 = 1

ffextend(a, P, v)

Extend the field K of definition of a by a root of the polynomial $P \in K[X]$ assumed to be irreducible over K. Return [r, m] where r is a root of P in the extension field L and m is a map from K to L, see ffmap. If v is given, the variable name is used to display the generator of L, else the name of the variable of P is used. A generator of L can be recovered using b = ffgen(r). The image of P in L[X] can be recovered using PL = ffmap(m, P).

```
? a = ffgen([3,5],'a);
? P = x^2-a; polisirreducible(P)
%2 = 1
? [r,m] = ffextend(a, P, 'b);
? r
%3 = b^9+2*b^8+b^7+2*b^6+b^4+1
? subst(ffmap(m, P), x, r)
%4 = 0
? ffgen(r)
%5 = b
```

fffrobenius(m, n)

Return the n-th power of the Frobenius map over the field of definition of m.

```
? a = ffgen([3,5],'a);
? f = fffrobenius(a);
? ffmap(f,a) == a^3
%3 = 1
? g = fffrobenius(a, 5);
? ffmap(g,a) == a
%5 = 1
? h = fffrobenius(a, 2);
? h == ffcompomap(f,f)
%7 = 1
```

ffgen(k, v)

Return a generator for the finite field k as a t_FFELT. The field k can be given by

- its order q
- the pair [p, f] where $q = p^f$
- a monic irreducible polynomial with t_INTMOD coefficients modulo a prime.
- a t_FFELT belonging to k.

If v is given, the variable name is used to display g, else the variable of the polynomial or the t_FFELT is used, else x is used.

When only the order is specified, the function uses the polynomial generated by ffinit and is deterministic: two calls to the function with the same parameters will always give the same generator.

For efficiency, the characteristic is not checked to be prime; similarly if a polynomial is given, we do not check whether it is irreducible.

To obtain a multiplicative generator, call ffprimroot on the result.

```
? g = ffgen(16, 't);
? g.mod \\ recover the underlying polynomial.
%2 = t^4+t^3+t^2+t+1
? g.pol \\ lift g as a t_POL
%3 = t
? g.p \\ recover the characteristic
%4 = 2
? fforder(g) \\ g is not a multiplicative generator
%5 = 5
? a = ffprimroot(g) \\ recover a multiplicative generator
%6 = t^3+t^2+t
? fforder(a)
%7 = 15
```

ffinit(*p*, *n*, *v*)

Computes a monic polynomial of degree n which is irreducible over \mathbb{F}_p , where p is assumed to be prime. This function uses a fast variant of Adleman and Lenstra's algorithm.

It is useful in conjunction with ffgen; for instance if P = ffinit(3,2), you can represent elements in \mathbb{F}_{3^2} in term of g = ffgen(P, 't). This can be abbreviated as $g = ffgen(3^2, 't)$, where the defining polynomial P can be later recovered as g.mod.

ffinvmap(m)

m being a map from *K* to *L* two finite fields, return the partial map *p* from *L* to *K* such that for all $k \in K$, p(m(k)) = k.

```
? a = ffgen([3,5],'a);
? b = ffgen([3,10],'b);
? m = ffembed(a, b);
? p = ffinvmap(m);
? u = random(a);
? v = ffmap(m, u);
? ffmap(p, v^2+v+2) == u^2+u+2
%7 = 1
? ffmap(p, b)
%8 = []
```

fflog(x, g, o)

Discrete logarithm of the finite field element x in base g, i.e. an e in \mathbb{Z} such that $g^e = o$. If present, o represents the multiplicative order of g, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the order of g. It may be set as a side effect of calling ffprimroot. The result is undefined if e does not exist. This function uses

- a combination of generic discrete log algorithms (see znlog)
- a cubic sieve index calculus algorithm for large fields of degree at least 5.
- Coppersmith's algorithm for fields of characteristic at most 5.

```
? t = ffgen(ffinit(7,5));
? o = fforder(t)
%2 = 5602 \\ not a primitive root.
? fflog(t^10,t)
%3 = 10
? fflog(t^10,t, o)
%4 = 10
? g = ffprimroot(t, &o);
? o \\ order is 16806, bundled with its factorization matrix
%6 = [16806, [2, 1; 3, 1; 2801, 1]]
? fforder(g, o)
%7 = 16806
? fflog(g^10000, g, o)
%8 = 10000
```

ffmap(m, x)

Given a (partial) map m between two finite fields, return the image of x by m. The function is applied recursively to the component of vectors, matrices and polynomials. If m is a partial map that is not defined at x, return [].

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? P = x^2+a*x+1; ? Q = ffmap(m,P); ? ffmap(m,poldisc(P)) == poldisc(Q) %6 = 1

ffmaprel(m, x)

Given a (partial) map m between two finite fields, express x as an algebraic element over the codomain of m in a way which is compatible with m. The function is applied recursively to the component of vectors, matrices and polynomials.

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? mi= ffinvmap(m); ? R = ffmaprel(mi,b) %5 = Mod(b,b^2+(a+1)*b+(a^2+2*a+2))

In particular, this function can be used to compute the relative minimal polynomial, norm and trace:

```
? minpoly(R)
%6 = x^2+(a+1)*x+(a^2+2*a+2)
? trace(R)
%7 = 2*a+2
? norm(R)
%8 = a^2+2*a+2
```

ffnbirred(q, n, fl)

Computes the number of monic irreducible polynomials over \mathbb{F}_q of degree exactly n, (flag = 0 or omitted) or at most n (flag = 1).

fforder(x, o)

Multiplicative order of the finite field element x. If o is present, it represents a multiple of the order of the

element, see DLfun (in the PARI manual); the preferred format for this parameter is [N, factor(N)], where N is the cardinality of the multiplicative group of the underlying finite field.

```
? t = ffgen(ffinit(nextprime(10^8), 5));
? g = ffprimroot(t, &o); \\ o will be useful!
? fforder(g^1000000, o)
time = 0 ms.
%5 = 5000001750000245000017150000600250008403
? fforder(g^1000000)
time = 16 ms. \\ noticeably slower, same result of course
%6 = 5000001750000245000017150000600250008403
```

ffprimroot(x, o)

Return a primitive root of the multiplicative group of the definition field of the finite field element x (not necessarily the same as the field generated by x). If present, o is set to a vector [ord, fa], where ord is the order of the group and fa its factorization factor(ord). This last parameter is useful in fflog and fforder, see DLfun (in the PARI manual).

```
? t = ffgen(ffinit(nextprime(10^7), 5));
? g = ffprimroot(t, &o);
? o[1]
\%3 = 100000950003610006859006516052476098
? o[2]
%4 =
[2 1]
[7 2]
[31 1]
[41 1]
[67 1]
[1523 1]
[10498781 1]
[15992881 1]
[46858913131 1]
? fflog(g^1000000, g, o)
time = 1,312 ms.
\%5 = 1000000
```

fft(w, P)

Let $w = [1, z, ..., z^{N-1}]$ from some primitive N-roots of unity z where N is a power of 2, and P be a polynomial < N, return the unnormalized discrete Fourier transform of P, $P(w[i]), 1 \le i \le N$. Also allow P to be a vector $[p_0, ..., p_n]$ representing the polynomial $\sum p_i X^i$. Composing fft and fftinv returns N times the original input coefficients.

? $w = rootsof1(4); fft(w, x^3+x+1)$

```
%1 = [3, 1, -1, 1]
? fftinv(w, %)
%2 = [4, 4, 0, 4]
? Polrev(%) / 4
%3 = x^3 + x + 1
? w = powers(znprimroot(5),3); fft(w, x^3+x+1)
%4 = [Mod(3,5),Mod(1,5),Mod(4,5),Mod(1,5)]
? fftinv(w, %)
%5 = [Mod(4,5),Mod(4,5),Mod(0,5),Mod(4,5)]
```

fftinv(w, P)

Let $w = [1, z, ..., z^{N-1}]$ from some primitive N-roots of unity z where N is a power of 2, and P be a polynomial < N, return the unnormalized discrete Fourier transform of P, $P(1/w[i]), 1 \le i \le N$. Also allow P to be a vector $[p_0, ..., p_n]$ representing the polynomial $\sum p_i X^i$. Composing fft and fftinv returns N times the original input coefficients.

```
? w = rootsof1(4); fft(w, x^3+x+1)
%1 = [3, 1, -1, 1]
? fftinv(w, %)
%2 = [4, 4, 0, 4]
? Polrev(%) / 4
%3 = x^3 + x + 1
? N = 512; w = rootsof1(N); T = random(1000 * x^(N-1));
? U = fft(w, T);
time = 3 ms.
? V = vector(N, i, subst(T, 'x, w[i]));
time = 65 ms.
? exponent(V - U)
%7 = -97
? round(Polrev(fftinv(w,U) / N)) == T
%8 = 1
```

fibonacci(x)

x - th Fibonacci number.

fileclose(n)

Close the file descriptor *n*, created via fileopen or fileextern. Finitely many files can be opened at a given time, closing them recycles file descriptors and avoids running out of them:

This is a limitation of the operating system and does not depend on PARI: if you open too many files in gp without closing them, the operating system will also prevent unrelated applications from opening files. Independently, your operating system (e.g. Windows) may prevent other applications from accessing or deleting your file while it is opened by gp. Quitting gp implicitly calls this function on all opened file descriptors.

On files opened for writing, this function also forces a write of all buffered data to the file system and completes all pending write operations. This function is implicitly called for all open file descriptors when exiting gp but it is cleaner and safer to call it explicitly, for instance in case of a gp crash or general system failure, which could cause data loss.

```
? n = fileopen("./here");
? while(l = fileread(n), print(l));
? fileclose(n);
? n = fileopen("./there", "w");
? for (i = 1, 100, filewrite(n, i^2+1))
? fileclose(n)
```

Until a fileclose, there is no guarantee that the file on disk contains all the expected data from previous filewrite s. (And even then the operating system may delay the actual write to hardware.)

Closing a file twice raises an exception:

```
? n = fileopen("/tmp/test");
? fileclose(n)
? fileclose(n)
*** at top-level: fileclose(n)
*** ^------
*** fileclose: invalid file descriptor 0
```

fileextern(str)

The string *str* is the name of an external command, i.e. one you would type from your UNIX shell prompt. This command is immediately run and the function returns a file descriptor attached to the command output as if it were read from a file.

```
? n = fileextern("ls -l");
? while(l = filereadstr(n), print(l))
? fileclose(n)
```

If the secure default is set, this function will raise en exception.

fileflush(n)

Flushes the file descriptor n, created via fileopen or fileextern. On files opened for writing, this function forces a write of all buffered data to the file system and completes all pending write operations. This function is implicitly called by fileclose but you may want to call it explicitly at synchronization points, for instance after writing a large result to file and before printing diagnostics on screen. (In order to be sure that the file contains the expected content on inspection.)

If n is omitted, flush all descriptors to output streams.

```
? n = fileopen("./here", "w");
? for (i = 1, 10^5, \
filewrite(n, i^2+1); \
if (i % 10000 == 0, fileflush(n)))
```

Until a fileflush or fileclose, there is no guarantee that the file contains all the expected data from previous filewrite s.

fileopen(path, mode)

Open the file pointed to by 'path' and return a file descriptor which can be used with other file functions.

The mode can be

- "r" (default): open for reading; allow fileread and filereadstr.
- "w": open for writing, discarding existing content; allow filewrite, filewrite1.
- "a": open for writing, appending to existing content; same operations allowed as "w".

Eventually, the file should be closed and the descriptor recycled using fileclose.

```
? n = fileopen("./here"); \\ "r" by default
? while (l = filereadstr(n), print(l)) \\ print successive lines
? fileclose(n) \\ done
```

In *read* mode, raise an exception if the file does not exist or the user does not have read permission. In *write* mode, raise an exception if the file cannot be written to. Trying to read or write to a file that was not opend with the right mode raises an exception.

```
? n = fileopen("./read", "r");
? filewrite(n, "test") \\ not open for writing
 *** at top-level: filewrite(n,"test")
 *** ^------
 *** filewrite: invalid file descriptor 0
```

fileread(n)

Read a logical line from the file attached to the descriptor n, opened for reading with fileopen. Return 0 at end of file.

A logical line is a full command as it is prepared by gp's preprocessor (skipping blanks and comments or assembling multiline commands between braces) before being fed to the interpreter. The function filereadstr would read a *raw* line exactly as input, up to the next carriage return n.

Compare raw lines

```
? n = fileopen("examples/bench.gp");
? while(l = filereadstr(n), print(l));
{
    u=v=p=q=1;
    for (k=1, 2000,
    [u,v] = [v,u+v];
    p *= v; q = lcm(q,v);
    if (k%50 == 0,
    print(k, " ", log(p)/log(q))
    )
}
```

and logical lines

```
? n = fileopen("examples/bench.gp");
? while(l = fileread(n), print(l));
u=v=p=q=1;for(k=1,2000,[u,v]=[v,u+v];p*=v;q=lcm(q,v);[...]
```

filereadstr(n)

Read a raw line from the file attached to the descriptor n, opened for reading with fileopen, discarding the terminating newline. In other words the line is read exactly as input, up to the next carriage return n. By comparison, fileread would read a logical line, as assembled by gp's preprocessor (skipping blanks and comments for instance).
filewrite(n, s)

Write the string s to the file attached to descriptor n, ending with a newline. The file must have been opened with fileopen in "w" or "a" mode. There is no guarantee that s is completely written to disk until fileclose:math:`(n)` is executed, which is automatic when quitting gp.

If the newline is not desired, use filewrite1.

Variant. The high-level function write is expensive when many consecutive writes are expected because it cannot use buffering. The low-level interface fileopen / filewrite / fileclose is more efficient:

```
? f = "/tmp/bigfile";
? for (i = 1, 10^5, write(f, i^2+1))
time = 240 ms.
? v = vector(10^5, i, i^2+1);
time = 10 ms. \\ computing the values is fast
? write("/tmp/bigfile2",v)
time = 12 ms. \\ writing them in one operation is fast
? n = fileopen("/tmp/bigfile", "w");
? for (i = 1, 10^5, filewrite(n, i^2+1))
time = 24 ms. \\ low-level write is ten times faster
? fileclose(n);
```

In the final example, the file needs not be in a consistent state until the ending fileclose is evaluated, e.g. some lines might be half-written or not present at all even though the corresponding filewrite was executed already. Both a single high-level write and a succession of low-level filewrite s achieve the same efficiency, but the latter is often more natural. In fact, concatenating naively the entries to be written is quadratic in the number of entries, hence much more expensive than the original write operations:

```
? v = []; for (i = 1, 10^5, v = concat(v,i))
time = 1min, 41,456 ms.
```

filewrite1(n, s)

Write the string s to the file attached to descriptor n. The file must have been opened with fileopen in "w" or "a" mode.

If you want to append a newline at the end of s, you can use $Str(s, "\n")$ or filewrite.

floor(x)

Floor of x. When x is in \mathbb{R} , the result is the largest integer smaller than or equal to x. Applied to a rational function, floor(x) returns the Euclidean quotient of the numerator by the denominator.

fold(*f*, *A*)

Apply the t_CLOSURE f of arity 2 to the entries of A, in order to return $f(\ldots f(A[1], A[2]), A[3]) \ldots$, A[#A]).

```
? fold((x,y)->x*y, [1,2,3,4])
%1 = 24
? fold((x,y)->[x,y], [1,2,3,4])
%2 = [[[1, 2], 3], 4]
? fold((x,f)->f(x), [2,sqr,sqr,sqr])
%3 = 256
? fold((x,y)->(x+y)/(1-x*y),[1..5])
%4 = -9/19
```

? bestappr(tan(sum(i=1,5,atan(i))))
%5 = -9/19

frac(x)

Fractional part of x. Identical to x - floor(x). If x is real, the result is in [0, 1].

fromdigits(x, b)

Gives the integer formed by the elements of x seen as the digits of a number in base b (b = 10 by default). This is the reverse of digits:

? digits(1234,5)
%1 = [1,4,4,1,4]
? fromdigits([1,4,4,1,4],5)
%2 = 1234

By convention, 0 has no digits:

```
? fromdigits([])
%3 = 0
```

galoischardet(gal, chi, o)

Let G be the group attached to the galoisinit structure gal, and let χ be the character of some representation ρ of the group G, where a polynomial variable is to be interpreted as an o-th root of 1. For instance, if [T,o] = galoischartable(gal) the characters χ are input as the columns of T.

Return the degree-1 character det ρ as the list of det $\rho(g)$, where g runs through representatives of the conjugacy classes in galoisconjclasses(gal), with the same ordering.

```
? P = x^5 - x^4 - 5*x^3 + 4*x^2 + 3*x - 1;
? polgalois(P)
%2 = [10, 1, 1, "D(5) = 5:2"]
? K = nfsplitting(P);
? gal = galoisinit(K); \\ dihedral of order 10
? [T,o] = galoischartable(gal);
? chi = T[,1]; \\ trivial character
? galoischardet(gal, chi, o)
%7 = [1, 1, 1, 1]~
? [galoischardet(gal, T[,i], o) | i <- [1..#T]] \\ all characters
%8 = [[1, 1, 1, 1]~, [1, 1, -1, 1]~, [1, 1, -1, 1]~]
```

galoischarpoly(gal, chi, o)

Let G be the group attached to the galoisinit structure gal, and let χ be the character of some representation ρ of the group G, where a polynomial variable is to be interpreted as an o-th root of 1, e.g., if [T,o] = galoischartable(gal) and χ is a column of T. Return the list of characteristic polynomials det $(1 - \rho(g)T)$, where g runs through representatives of the conjugacy classes in galoisconjclasses(gal), with the same ordering.

```
? T = x^5 - x^4 - 5*x^3 + 4*x^2 + 3*x - 1;
? polgalois(T)
%2 = [10, 1, 1, "D(5) = 5:2"]
? K = nfsplitting(T);
? gal = galoisinit(K); \\ dihedral of order 10
? [T,o] = galoischartable(gal);
```

```
? o
%5 = 5
? galoischarpoly(gal, T[,1], o) \\ T[,1] is the trivial character
%6 = [-x + 1, -x + 1, -x + 1]~
? galoischarpoly(gal, T[,3], o)
%7 = [x^2 - 2*x + 1,
x^2 + (y^3 + y^2 + 1)*x + 1,
-x^2 + 1,
x^2 + (-y^3 - y^2)*x + 1]~
```

galoischartable(gal)

Compute the character table of G, where G is the underlying group of the galoisinit structure *gal*. The input *gal* is also allowed to be a t_VEC of permutations that is closed under products. Let N be the number of conjugacy classes of G. Return a t_VEC [M, e] where $e \ge 1$ is an integer and M is a square t_MAT of size N giving the character table of G.

- Each column corresponds to an irreducible character; the characters are ordered by increasing dimension and the first column is the trivial character (hence contains only 1's).
- Each row corresponds to a conjugacy class; the conjugacy classes are ordered as specified by galoisconjclasses(gal), in particular the first row corresponds to the identity and gives the dimension $\chi(1)$ of the irreducible representation attached to the successive characters χ .

The value M[i, j] of the character j at the conjugacy class i is represented by a polynomial in y whose variable should be interpreted as an e-th root of unity, i.e. as the lift of

Mod(y, polcyclo(e,'y))

(Note that M is the transpose of the usual orientation for character tables.)

The integer e divides the exponent of the group G and is chosen as small as possible; for instance e = 1 when the characters are all defined over \mathbb{Q} , as is the case for S_n . Examples:

```
? K = nfsplitting(x^4+x+1);
? gal = galoisinit(K);
? [M,e] = galoischartable(gal);
? M \sim \setminus  take the transpose to get the usual orientation
%4 =
[1 \ 1 \ 1 \ 1 \ 1]
[1 - 1 - 1 1 1]
[2 \ 0 \ 0 \ -1 \ 2]
[3 -1 1 0 -1]
[3 1 -1 0 -1]
?е
\%5 = 1
? {G = [Vecsmall([1, 2, 3, 4, 5]), Vecsmall([1, 5, 4, 3, 2]),
Vecsmall([2, 1, 5, 4, 3]), Vecsmall([2, 3, 4, 5, 1]),
Vecsmall([3, 2, 1, 5, 4]), Vecsmall([3, 4, 5, 1, 2]),
Vecsmall([4, 3, 2, 1, 5]), Vecsmall([4, 5, 1, 2, 3]),
Vecsmall([5, 1, 2, 3, 4]), Vecsmall([5, 4, 3, 2, 1])];}
```

```
\\G = D10
? [M,e] = galoischartable(G);
? M~
%8 =
[1 1 1 1]
[1 -1 1 1]
[2 0 -y^3 - y^2 - 1 y^3 + y^2]
[2 0 y^3 + y^2 -y^3 - y^2 - 1]
? e
%9 = 5
```

galoisconjclasses(gal)

gal being output by galoisinit, return the list of conjugacy classes of the underlying group. The ordering of the classes is consistent with galoischartable and the trivial class comes first.

```
? G = galoisinit(x^6+108);
? galoisidentify(G)
%2 = [6, 1] \\ S_3
? S = galoisconjclasses(G)
%3 = [[Vecsmall([1,2,3,4,5,6])],
[Vecsmall([3,1,2,6,4,5]),Vecsmall([2,3,1,5,6,4])],
[Vecsmall([6,5,4,3,2,1]),Vecsmall([5,4,6,2,1,3]),
Vecsmall([4,6,5,1,3,2])]]
? [[permorder(c[1]),#c] | c <- S ]
%4 = [[1,1], [3,2], [2,3]]
```

This command also accepts subgroups returned by galoissubgroups:

```
? subs = galoissubgroups(G); H = subs[5];
? galoisidentify(H)
%2 = [2, 1] \\ Z/2
? S = galoisconjclasses(subgroups_of_G[5]);
? [[permorder(c[1]),#c] | c <- S ]
%4 = [[1,1], [2,1]]
```

galoisexport(gal, flag)

gal being be a Galois group as output by galoisinit, export the underlying permutation group as a string suitable for (no flags or flag = 0) GAP or (flag = 1) Magma. The following example compute the index of the underlying abstract group in the GAP library:

```
? G = galoisinit(x^6+108);
? s = galoisexport(G)
%2 = "Group((1, 2, 3)(4, 5, 6), (1, 4)(2, 6)(3, 5))"
? extern("echo \"IdGroup("s");\" | gap -q")
%3 = [6, 1]
? galoisidentify(G)
%4 = [6, 1]
```

This command also accepts subgroups returned by galoissubgroups.

To *import* a GAP permutation into gp (for galoissubfields for instance), the following GAP function may be useful:

PermToGP := function(p, n)
return Permuted([1..n],p);
end;
gap> p:= (1,26)(2,5)(3,17)(4,32)(6,9)(7,11)(8,24)(10,13)(12,15)(14,27)
(16,22)(18,28)(19,20)(21,29)(23,31)(25,30)
gap> PermToGP(p,32);
[26, 5, 17, 32, 2, 9, 11, 24, 6, 13, 7, 15, 10, 27, 12, 22, 3, 28, 20, 19,
29, 16, 31, 8, 30, 1, 14, 18, 21, 25, 23, 4]

galoisfixedfield(gal, perm, flag, v)

gal being be a Galois group as output by galoisinit and *perm* an element of *gal.group*, a vector of such elements or a subgroup of *gal* as returned by galoissubgroups, computes the fixed field of *gal* by the automorphism defined by the permutations *perm* of the roots *gal.roots*. P is guaranteed to be squarefree modulo *gal.p*.

If no flags or flag = 0, output format is the same as for nfsubfield, returning [P, x] such that P is a polynomial defining the fixed field, and x is a root of P expressed as a polmod in *gal.pol*.

If flag = 1 return only the polynomial P.

If flag = 2 return [P, x, F] where P and x are as above and F is the factorization of gal.pol over the field defined by P, where variable v (y by default) stands for a root of P. The priority of v must be less than the priority of the variable of gal.pol (see priority (in the PARI manual)). In this case, P is also expressed in the variable v for compatibility with F. Example:

? G = galoisinit(x⁴+1); ? galoisfixedfield(G,G.group[2],2) %2 = [y² - 2, Mod(- x³ + x, x⁴ + 1), [x² - y^{*}x + 1, x² + y^{*}x + 1]]

computes the factorization $x^4 + 1 = (x^2 - \sqrt{2}x + 1)(x^2 + \sqrt{2}x + 1)$

galoisgetgroup(a, b)

Query the galpol package for a group of order a with index b in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien.

The current version of galpol supports groups of order $a \le 143$. If b is omitted, return the number of isomorphism classes of groups of order a.

galoisgetname(a, b)

Query the galpol package for a string describing the group of order a with index b in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien. The strings were generated using the GAP4 function StructureDescription. The command below outputs the names of all abstract groups of order 12:

```
? o = 12; N = galoisgetgroup(o); \\ # of abstract groups of order 12
? for(i=1, N, print(i, ". ", galoisgetname(o,i)))
1. C3 : C4
2. C12
3. A4
4. D12
5. C6 x C2
```

The current version of galpol supports groups of order $a \le 143$. For $a \ge 16$, it is possible for different groups to have the same name:

```
? o = 20; N = galoisgetgroup(o);
? for(i=1, N, print(i, ". ", galoisgetname(o,i)))
1. C5 : C4
2. C20
3. C5 : C4
4. D20
5. C10 x C2
```

galoisgetpol(a, b, s)

Query the galpol package for a polynomial with Galois group isomorphic to GAP4(a,b), totally real if s = 1 (default) and totally complex if s = 2. The current version of galpol supports groups of order $a \le 143$. The output is a vector [pol, den] where

- pol is the polynomial of degree a
- den is the denominator of nfgaloisconj(pol). Pass it as an optional argument to galoisinit or nfgaloisconj to speed them up:

```
? [pol,den] = galoisgetpol(64,4,1);
? G = galoisinit(pol);
time = 352ms
? galoisinit(pol, den); \\ passing 'den' speeds up the computation
time = 264ms
? % == %`
%4 = 1 \\ same answer
```

If b and s are omitted, return the number of isomorphism classes of groups of order a.

galoisidentify(gal)

gal being be a Galois group as output by galoisinit, output the isomorphism class of the underlying abstract group as a two-components vector [o, i], where o is the group order, and i is the group index in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien.

This command also accepts subgroups returned by galoissubgroups.

The current implementation is limited to degree less or equal to 127. Some larger "easy" orders are also supported.

The output is similar to the output of the function IdGroup in GAP4. Note that GAP4 IdGroup handles all groups of order less than 2000 except 1024, so you can use galoisexport and GAP4 to identify large Galois groups.

galoisinit(pol, den)

Computes the Galois group and all necessary information for computing the fixed fields of the Galois extension K/\mathbb{Q} where K is the number field defined by *pol* (monic irreducible polynomial in $\mathbb{Z}[X]$ or a number field as output by nfinit). The extension K/\mathbb{Q} must be Galois with Galois group "weakly" super-solvable, see below; returns 0 otherwise. Hence this permits to quickly check whether a polynomial of order strictly less than 48 is Galois or not.

The algorithm used is an improved version of the paper "An efficient algorithm for the computation of Galois automorphisms", Bill Allombert, Math. Comp, vol. 73, 245, 2001, pp. 359–375.

A group G is said to be "weakly" super-solvable if there exists a normal series

 $1=H_0 \triangleleft H_1 \triangleleft \ldots \triangleleft H_{n-1} \triangleleft H_n$

such that each H_i is normal in G and for i < n, each quotient group H_{i+1}/H_i is cyclic, and either $H_n = G$ (then G is super-solvable) or G/H_n is isomorphic to either A_4 , S_4 or the group (3x3) : 4 (GAP4(36,9)) then $[o_1, ..., o_g]$ ends by [3, 3, 4].

In practice, almost all small groups are WKSS, the exceptions having order 48(2), 56(1), 60(1), 72(3), 75(1), 80(1), 96(10), 112(1), 120(3) and >= 144.

This function is a prerequisite for most of the galoisxxx routines. For instance:

```
P = x^6 + 108;
G = galoisinit(P);
L = galoissubgroups(G);
vector(#L, i, galoisisabelian(L[i],1))
vector(#L, i, galoisidentify(L[i]))
```

The output is an 8-component vector gal.

gal[1] contains the polynomial pol (:emphasis:`gal.pol`).

gal[2] is a three-components vector [p, e, q] where p is a prime number (:emphasis:`gal.p`) such that pol totally split modulo p, e is an integer and $q = p^e$ (:emphasis:`gal.mod`) is the modulus of the roots in :emphasis:`gal.roots`.

gal[3] is a vector L containing the p-adic roots of pol as integers implicitly modulo :emphasis:`gal.mod`. (:emphasis:`gal.roots`).

gal[4] is the inverse of the Vandermonde matrix of the *p*-adic roots of *pol*, multiplied by gal[5].

gal[5] is a multiple of the least common denominator of the automorphisms expressed as polynomial in a root of pol.

gal[6] is the Galois group G expressed as a vector of permutations of L (:emphasis:`gal.group`).

gal[7] is a generating subset $S = [s_1, ..., s_g]$ of G expressed as a vector of permutations of L (:emphasis:`gal.gen`).

gal[8] contains the relative orders $[o_1, ..., o_q]$ of the generators of S (:emphasis:`gal.orders`).

Let H_n be as above, we have the following properties:

* if $G/H_n A_4$ then $[o_1, ..., o_q]$ ends by [2, 2, 3].

* if $G/H_n S_4$ then $[o_1, ..., o_g]$ ends by [2, 2, 3, 2].

* if $G/H_n(3x3): 4$ (GAP4(36,9)) then $[o_1, ..., o_g]$ ends by [3, 3, 4].

* for $1 \le i \le g$ the subgroup of G generated by $[s_1, ..., s_i]$ is normal, with the exception of i = g - 2 in the A_4 case and of i = g - 3 in the S_4 case.

* the relative order o_i of s_i is its order in the quotient group $G/\langle s_1, ..., s_{i-1} \rangle$, with the same exceptions.

* for any $x \in G$ there exists a unique family $[e_1, ..., e_q]$ such that (no exceptions):

- for $1 \le i \le g$ we have $0 \le e_i \le o_i$

$$-x = g_1^{e_1} g_2^{e_2} \dots g_n^{e_n}$$

If present den must be a suitable value for gal[5].

galoisisabelian(gal, flag)

gal being as output by galoisinit, return 0 if gal is not an abelian group, and the HNF matrix of gal over gal.gen if flag = 0, 1 if flag = 1, and the SNF matrix of gal if flag = 2.

This command also accepts subgroups returned by galoissubgroups.

galoisisnormal(gal, subgrp)

gal being as output by galoisinit, and *subgrp* a subgroup of *gal* as output by galoissubgroups, return 1 if *subgrp* is a normal subgroup of *gal*, else return 0.

This command also accepts subgroups returned by galoissubgroups.

galoispermtopol(gal, perm)

gal being a Galois group as output by galoisinit and perm a element of gal.group, return the polynomial defining the Galois automorphism, as output by nfgaloisconj, attached to the permutation perm of the roots gal.roots. perm can also be a vector or matrix, in this case, galoispermtopol is applied to all components recursively.

Note that

G = galoisinit(pol);
galoispermtopol(G, G[6])~

is equivalent to nfgaloisconj(pol), if degree of *pol* is greater or equal to 2.

galoissubcyclo(N, H, fl, v)

Computes the subextension of $\mathbb{Q}(\zeta_n)$ fixed by the subgroup $H \subset (\mathbb{Z}/n\mathbb{Z})^*$. By the Kronecker-Weber theorem, all abelian number fields can be generated in this way (uniquely if *n* is taken to be minimal).

The pair (n, H) is deduced from the parameters (N, H) as follows

- N an integer: then n = N; H is a generator, i.e. an integer or an integer modulo n; or a vector of generators.
- N the output of znstar(n) or znstar(n, 1). H as in the first case above, or a matrix, taken to be a HNF left divisor of the SNF for (Z/nZ)* (:math:`N.cyc`), giving the generators of H in terms of :math:`N.gen`.
- N the output of bnrinit(bnfinit(y), :math:`m)` where m is a module. H as in the first case, or a matrix taken to be a HNF left divisor of the SNF for the ray class group modulo m (of type :math:`N.cyc`), giving the generators of H in terms of :math:`N.bid.gen` (= :math:`N`.gen if N includes generators).

In this last case, beware that H is understood relatively to N; in particular, if the infinite place does not divide the module, e.g if m is an integer, then it is not a subgroup of $(\mathbb{Z}/n\mathbb{Z})^*$, but of its quotient by 1.

If fl = 0, compute a polynomial (in the variable v) defining the subfield of $\mathbb{Q}(\zeta_n)$ fixed by the subgroup H of $(\mathbb{Z}/n\mathbb{Z})^*$.

If fl = 1, compute only the conductor of the abelian extension, as a module.

If fl = 2, output [pol, N], where pol is the polynomial as output when fl = 0 and N the conductor as output when fl = 1.

The following function can be used to compute all subfields of $\mathbb{Q}(\zeta_n)$ (of exact degree d, if d is set):

```
subcyclo(n, d = -1)=
{ my(bnr,L,IndexBound);
IndexBound = if (d < 0, n, [d]);
bnr = bnrinit(bnfinit(y), [n,[1]]);
L = subgrouplist(bnr, IndexBound, 1);
vector(#L,i, galoissubcyclo(bnr,L[i]));
}</pre>
```

Setting L = subgrouplist(bnr, IndexBound) would produce subfields of exact conductor noo.

galoissubfields(G, flag, v)

Outputs all the subfields of the Galois group G, as a vector. This works by applying galoisfixedfield to all subgroups. The meaning of *flag* is the same as for galoisfixedfield.

galoissubgroups(G)

Outputs all the subgroups of the Galois group gal. A subgroup is a vector [gen, orders], with the same meaning as for gal.gen and gal.orders. Hence gen is a vector of permutations generating the subgroup, and orders is the

relatives orders of the generators. The cardinality of a subgroup is the product of the relative orders. Such subgroup can be used instead of a Galois group in the following command: galoisisabelian, galoissubgroups, galoisexport and galoisidentify.

To get the subfield fixed by a subgroup sub of gal, use

```
galoisfixedfield(gal,sub[1])
```

gamma(s, precision)

For s a complex number, evaluates Euler's gamma function

$$\Gamma(s) = \int_0^o ot^{s-1} \exp(-t) dt.$$

Error if s is a nonpositive integer, where Γ has a pole.

For s a t_PADIC, evaluates the Morita gamma function at s, that is the unique continuous p-adic function on the p-adic integers extending $\Gamma_p(k) = (-1)^k \prod_{i < k}^{\prime} j$, where the prime means that p does not divide j.

```
? gamma(1/4 + 0(5^10))
%1= 1 + 4*5 + 3*5^4 + 5^6 + 5^7 + 4*5^9 + 0(5^10)
? algdep(%,4)
%2 = x^4 + 4*x^2 + 5
```

gammah(x, precision)

Gamma function evaluated at the argument x + 1/2.

gammamellininv(*G*, *t*, *m*, *precision*)

Returns the value at t of the inverse Mellin transform G initialized by gammamellininvinit. If the optional parameter m is present, return the m-th derivative $G^{(m)}(t)$.

? G = gammamellininvinit([0]); ? gammamellininv(G, 2) - 2*exp(-Pi*2^2) %2 = -4.484155085839414627 E-44

The shortcut

```
gammamellininv(A,t,m)
```

for

```
gammamellininv(gammamellininvinit(A,m), t)
```

is available.

gammamellininvasymp(A, serprec, n)

Return the first n terms of the asymptotic expansion at infinity of the m-th derivative $K^{(m)}(t)$ of the inverse Mellin transform of the function

$$f(s) = \Gamma_{\mathbb{R}}(s+a_1)...\Gamma_{\mathbb{R}}(s+a_d),$$

where A is the vector $[a_1, ..., a_d]$ and $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2}\Gamma(s/2)$ (Euler's gamma). The result is a vector [M[1]...M[n]] with M[1] = 1, such that

$$K^{(m)}(t) = \sqrt{2^{d+1}/dt} t^{a+m(2/d-1)} e^{-d\pi t^{2/d}} \sum_{n>=0} M[n+1](\pi t^{2/d})^{-n}$$

with $a = (1 - d + \sum_{1 \le i \le d} a_i)/d$. We also allow A to be the output of gammamellininvinit.

gammamellininvinit(A, m, precision)

Initialize data for the computation by gammamellininv of the m-th derivative of the inverse Mellin transform of the function

$$f(s) = \Gamma_{\mathbb{R}}(s+a_1)...\Gamma_{\mathbb{R}}(s+a_d)$$

where A is the vector $[a_1, ..., a_d]$ and $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2}\Gamma(s/2)$ (Euler's gamma). This is the special case of Meijer's G functions used to compute L-values via the approximate functional equation. By extension, A is allowed to be an Ldata or an Linit, understood as the inverse Mellin transform of the L-function gamma factor.

Caveat. Contrary to the PARI convention, this function guarantees an *absolute* (rather than relative) error bound.

For instance, the inverse Mellin transform of $\Gamma_{\mathbb{R}}(s)$ is $2 \exp(-\pi z^2)$:

```
? G = gammamellininvinit([0]);
? gammamellininv(G, 2) - 2*exp(-Pi*2^2)
%2 = -4.484155085839414627 E-44
```

The inverse Mellin transform of $\Gamma_{\mathbb{R}}(s+1)$ is $2z \exp(-\pi z^2)$, and its second derivative is $4\pi z \exp(-\pi z^2)(2\pi z^2 - 3)$:

? G = gammamellininvinit([1], 2); ? a(z) = 4*Pi*z*exp(-Pi*z^2)*(2*Pi*z^2-3); ? b(z) = gammamellininv(G,z); ? t(z) = b(z) - a(z); ? t(3/2) %3 = -1.4693679385278593850 E-39

gcd(*x*, *y*)

Creates the greatest common divisor of x and y. If you also need the u and v such that x * u + y * v = gcd(x, y), use the gcdext function. x and y can have rather quite general types, for instance both rational numbers. If y is omitted and x is a vector, returns the gcd of all components of x, i.e. this is equivalent to content(x).

When x and y are both given and one of them is a vector/matrix type, the GCD is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to gcd(x, y[i]), resp. gcd(x, y[,i]). Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, gcd is not commutative.

The algorithm used is a naive Euclid except for the following inputs:

- integers: use modified right-shift binary ("plus-minus" variant).
- univariate polynomials with coefficients in the same number field (in particular rational): use modular gcd algorithm.
- general polynomials: use the subresultant algorithm if coefficient explosion is likely (non modular coefficients).

If u and v are polynomials in the same variable with *inexact* coefficients, their gcd is defined to be scalar, so that

```
? a = x + 0.0; gcd(a,a)
%1 = 1
? b = y*x + 0(y); gcd(b,b)
%2 = y
? c = 4*x + 0(2^3); gcd(c,c)
%3 = 4
```

A good quantitative check to decide whether such a gcd "should be" nontrivial, is to use polresultant: a value close to 0 means that a small deformation of the inputs has nontrivial gcd. You may also use gcdext, which does try to compute an approximate gcd d and provides u, v to check whether ux + vy is close to d.

gcdext(x, y)

Returns [u, v, d] such that d is the gcd of x, y, x * u + y * v = gcd(x, y), and u and v minimal in a natural sense. The arguments must be integers or polynomials.

? [u, v, d] = gcdext(32,102) %1 = [16, -5, 2] ? d %2 = 2 ? gcdext(x^2-x, x^2+x-2) %3 = [-1/2, 1/2, x - 1]

If x, y are polynomials in the same variable and *inexact* coefficients, then compute u, v, d such that x * u + y * v = d, where d approximately divides both and x and y; in particular, we do not obtain gcd(x,y) which is *defined* to be a scalar in this case:

? a = x + 0.0; gcd(a,a) %1 = 1 ? gcdext(a,a) %2 = [0, 1, x + 0.E-28] ? gcdext(x-Pi, 6*x^2-zeta(2)) %3 = [-6*x - 18.8495559, 1, 57.5726923]

For inexact inputs, the output is thus not well defined mathematically, but you obtain explicit polynomials to check whether the approximation is close enough for your needs.

genus2red(PQ, p)

Let PQ be a polynomial P, resp. a vector [P, Q] of polynomials, with rational coefficients. Determines the reduction at p > 2 of the (proper, smooth) genus 2 curve C/\mathbb{Q} , defined by the hyperelliptic equation $y^2 = P(x)$, resp. $y^2 + Q(x) * y = P(x)$. (The special fiber X_p of the minimal regular model X of C over \mathbb{Z} .)

If p is omitted, determines the reduction type for all (odd) prime divisors of the discriminant.

This function was rewritten from an implementation of Liu's algorithm by Cohen and Liu (1994), genus2reduction-0.3, see http://www.math.u-bordeaux.fr/~liu/G2R/.

CAVEAT. The function interface may change: for the time being, it returns [N, FaN, T, V] where N is either the local conductor at p or the global conductor, FaN is its factorization, $y^2 = T$ defines a minimal model over $\mathbb{Z}[1/2]$ and V describes the reduction type at the various considered p. Unfortunately, the program is not complete for p = 2, and we may return the odd part of the conductor only: this is the case if the factorization includes the (impossible) term 2^{-1} ; if the factorization contains another power of 2, then this is the exact local conductor at 2 and N is the global conductor.

```
? default(debuglevel, 1);
? genus2red(x^6 + 3*x^3 + 63, 3)
(potential) stable reduction: [1, []]
reduction at p: [III{9}] page 184, [3, 3], f = 10
%1 = [59049, Mat([3, 10]), x^6 + 3*x^3 + 63, [3, [1, []],
["[III{9}] page 184", [3, 3]]]]
? [N, FaN, T, V] = genus2red(x^3-x^2-1, x^2-x); \\ X_1(13), global reduction
p = 13
```

```
(potential) stable reduction: [5, [Mod(0, 13), Mod(0, 13)]]
reduction at p: [I{0}-II-0] page 159, [], f = 2
? N
%3 = 169
? FaN
%4 = Mat([13, 2]) \\ in particular, good reduction at 2 !
? T
%5 = x^6 + 58*x^5 + 1401*x^4 + 18038*x^3 + 130546*x^2 + 503516*x + 808561
? V
%6 = [[13, [5, [Mod(0, 13), Mod(0, 13)]], ["[I{0}-II-0] page 159", []]]]
```

We now first describe the format of the vector $V = V_p$ in the case where p was specified (local reduction at p): it is a triple [p, stable, red]. The component stable = [type, vecj] contains information about the stable reduction after a field extension; depending on type s, the stable reduction is

- 1: smooth (i.e. the curve has potentially good reduction). The Jacobian J(C) has potentially good reduction.
- 2: an elliptic curve E with an ordinary double point; *vecj* contains $j \mod p$, the modular invariant of E. The (potential) semi-abelian reduction of J(C) is the extension of an elliptic curve (with modular invariant $j \mod p$) by a torus.
- 3: a projective line with two ordinary double points. The Jacobian J(C) has potentially multiplicative reduction.
- 4: the union of two projective lines crossing transversally at three points. The Jacobian J(C) has potentially multiplicative reduction.
- 5: the union of two elliptic curves E₁ and E₂ intersecting transversally at one point; vecj contains their modular invariants j₁ and j₂, which may live in a quadratic extension of F_p and need not be distinct. The Jacobian J(C) has potentially good reduction, isomorphic to the product of the reductions of E₁ and E₂.
- 6: the union of an elliptic curve *E* and a projective line which has an ordinary double point, and these two components intersect transversally at one point; *vecj* contains *j* mod *p*, the modular invariant of *E*. The (potential) semi-abelian reduction of *J*(*C*) is the extension of an elliptic curve (with modular invariant *j* mod *p*) by a torus.
- 7: as in type 6, but the two components are both singular. The Jacobian J(C) has potentially multiplicative reduction.

The component red = [NUtype, neron] contains two data concerning the reduction at p without any ramified field extension.

The *NUtype* is a t_STR describing the reduction at *p* of *C*, following Namikawa-Ueno, *The complete classification* of fibers in pencils of curves of genus two, Manuscripta Math., vol. 9, (1973), pages 143-186. The reduction symbol is followed by the corresponding page number or page range in this article.

The second datum *neron* is the group of connected components (over an algebraic closure of \mathbb{F}_p) of the Néron model of J(C), given as a finite abelian group (vector of elementary divisors).

If p = 2, the *red* component may be omitted altogether (and replaced by [], in the case where the program could not compute it. When p was not specified, V is the vector of all V_p , for all considered p.

Notes about Namikawa-Ueno types.

- A lower index is denoted between braces: for instance, [I{2}-II-5] means [I_2-II-5].
- If K and K' are Kodaira symbols for singular fibers of elliptic curves, then [:math:`K-K'-m]` and [:math:`K'-K-m]` are the same.

We define a total ordering on Kodaira symbol by fixing I < I* < II < II*, ... If the reduction type is the same, we order by the number of components, e.g. $I_2 < I_4$, etc. Then we normalize our output so that K <= K'.

- [:math: K-K'--1] is [:math: $K-K'-\alpha$] in the notation of Namikawa-Ueno.
- The figure [2I_0-m] in Namikawa-Ueno, page 159, must be denoted by [2I_0-(m+1)].

getabstime()

Returns the CPU time (in milliseconds) elapsed since gp startup. This provides a reentrant version of gettime:

```
my (t = getabstime());
...
print("Time: ", strtime(getabstime() - t));
```

For a version giving wall-clock time, see getwalltime.

getcache()

Returns information about various auto-growing caches. For each resource, we report its name, its size, the number of cache misses (since the last extension), the largest cache miss and the size of the cache in bytes.

The caches are initially empty, then set automatically to a small inexpensive default value, then grow on demand up to some maximal value. Their size never decreases, they are only freed on exit.

The current caches are

- Hurwitz class numbers H(D) for $||D|| \le N$, computed in time $O(N^{3/2})$ using O(N) space.
- Factorizations of small integers up to N, computed in time $O(N^{1+\varepsilon})$ using $O(N \log N)$ space.
- Divisors of small integers up to N, computed in time $O(N^{1+\varepsilon})$ using $O(N \log N)$ space.
- Coredisc's of negative integers down to -N, computed in time $O(N^{1+\varepsilon})$ using O(N) space.
- Primitive dihedral forms of weight 1 and level up to N, computed in time $O(N^{2+\varepsilon})$ and space $O(N^2)$.

```
? getcache() \\ on startup, all caches are empty
%1 =
[ "Factors" 0 0 0 0]
[ "Divisors" 0 0 0 0]
[ "H" 0 0 0 0]
[ "CorediscF" 0 0 0 0]
[ "Dihedral" 0 0 0 0]
? mfdim([500,1,0],0); \\ nontrivial computation
time = 540 ms.
? getcache()
%3 =
[ "Factors" 50000 0 0 4479272]
["Divisors" 50000 1 100000 5189808]
[ "H" 50000 0 0 400008]
[ "Dihedral" 1000 0 0 2278208]
```

getenv(s)

Return the value of the environment variable s if it is defined, otherwise return 0.

getheap()

Returns a two-component row vector giving the number of objects on the heap and the amount of memory they occupy in long words. Useful mainly for debugging purposes.

getlocalbitprec(precision)

Returns the current dynamic bit precision.

getlocalprec(precision)

Returns the current dynamic precision, in decimal digits.

getrand()

Returns the current value of the seed used by the pseudo-random number generator random. Useful mainly for debugging purposes, to reproduce a specific chain of computations. The returned value is technical (reproduces an internal state array), and can only be used as an argument to setrand.

getstack()

Returns the current value of top - avma, i.e. the number of bytes used up to now on the stack. Useful mainly for debugging purposes.

gettime()

Returns the CPU time (in milliseconds) used since either the last call to gettime, or to the beginning of the containing GP instruction (if inside gp), whichever came last.

For a reentrant version, see getabstime.

For a version giving wall-clock time, see getwalltime.

getwalltime()

Returns the time (in milliseconds) elapsed since 00:00:00 UTC Thursday 1, January 1970 (the Unix epoch).

```
my (t = getwalltime());
...
print("Time: ", strtime(getwalltime() - t));
```

halfgcd(x, y)

Let inputs x and y be both integers, or both polynomials in the same variable. Return a vector [M, [a,b]~], where M is an invertible $2x^2$ matrix such that $M^*[x,y] \sim = [a,b] \sim$, where b is small. More precisely,

• polynomial case: $\det M$ has degree 0 and we have

 $\deg a \ge ceil\max(\deg x, \deg y))/2 > \deg b.$

• integer case: $\det M = 1$ and we have

 $Assuming: math: `x`and: math: `y`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q'are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `M^{-1}`has nonnegative, then: math: `M^{-1}`has nonne$

hammingweight(x)

If x is a t_INT, return the binary Hamming weight of ||x||. Otherwise x must be of type t_POL, t_VEC, t_COL, t_VECSMALL, or t_MAT and the function returns the number of nonzero coefficients of x.

```
? hammingweight(15)
%1 = 4
? hammingweight(x^100 + 2*x + 1)
%2 = 3
? hammingweight([Mod(1,2), 2, Mod(0,3)])
%3 = 2
? hammingweight(matid(100))
%4 = 100
```

hilbert(x, y, p)

Hilbert symbol of x and y modulo the prime p, p = 0 meaning the place at infinity (the result is undefined if p! = 0 is not prime).

It is possible to omit p, in which case we take p = 0 if both x and y are rational, or one of them is a real number. And take p = q if one of x, y is a t_INTMOD modulo q or a q-adic. (Incompatible types will raise an error.)

hyperellcharpoly(X)

X being a nonsingular hyperelliptic curve defined over a finite field, return the characteristic polynomial of the Frobenius automorphism. X can be given either by a squarefree polynomial P such that $X : y^2 = P(x)$ or by a vector [P, Q] such that $X : y^2 + Q(x)y = P(x)$ and $Q^2 + 4P$ is squarefree.

hyperellpadicfrobenius (Q, q, n)

Let X be the curve defined by $y^2 = Q(x)$, where Q is a polynomial of degree d over \mathbb{Q} and $q \ge d$ is a prime such that X has good reduction at q. Return the matrix of the Frobenius endomorphism φ on the crystalline module $D_p(X) = \mathbb{Q}_p \otimes H^1_{dR}(X/\mathbb{Q})$ with respect to the basis of the given model $(\omega, x\omega, ..., x^{g-1}\omega)$, where $\omega = dx/(2y)$ is the invariant differential, where g is the genus of X (either d = 2g + 1 or d = 2g + 2). The characteristic polynomial of φ is the numerator of the zeta-function of the reduction of the curve X modulo q. The matrix is computed to absolute q-adic precision q^n .

Alternatively, q may be of the form [T, p] where p is a prime, T is a polynomial with integral coefficients whose projection to $\mathbb{F}_p[t]$ is irreducible, X is defined over $K = \mathbb{Q}[t]/(T)$ and has good reduction to the finite field $\mathbb{F}_q = \mathbb{F}_p[t]/(T)$. The matrix of φ on $D_q(X) = \mathbb{Q}_q \otimes H^1_{dR}(X/K)$ is computed to absolute p-adic precision p^n .

```
? M=hyperellpadicfrobenius(x^5+'a*x+1,['a^2+1,3],10);
? liftall(M)
[48107*a + 38874 9222*a + 54290 41941*a + 8931 39672*a + 28651]
[ 21458*a + 4763 3652*a + 22205 31111*a + 42559 39834*a + 40207]
[ 13329*a + 4140 45270*a + 25803 1377*a + 32931 55980*a + 21267]
[15086*a + 26714 33424*a + 4898 41830*a + 48013 5913*a + 24088]
? centerlift(simplify(liftpol(charpoly(M))))
%8 = x^4+4*x^2+81
? hyperellcharpoly((x^5+Mod(a,a^2+1)*x+1)*Mod(1,3))
%9 = x^4+4*x^2+81
```

hyperellratpoints(X, h, flag)

X being a nonsingular hyperelliptic curve given by an rational model, return a vector containing the affine rational points on the curve of naive height less than h.a If flag = 1, stop as soon as a point is found; return either an empty vector or a vector containing a single point.

X is given either by a squarefree polynomial P such that $X : y^2 = P(x)$ or by a vector [P,Q] such that $X : y^2 + Q(x)y = P(x)$ and $Q^2 + 4P$ is squarefree.

The parameter h can be

- an integer H: find the points [n/d, y] whose abscissas x = n/d have naive height (= max(||n||, d)) less than H;
- a vector [N, D] with $D \le N$: find the points [n/d, y] with $||n|| \le N$, $d \le D$.
- a vector $[N, [D_1, D_2]]$ with $D_1 < D_2 <= N$ find the points [n/d, y] with $||n|| \le N$ and $D_1 \le d \le D_2$.

hypergeom(*N*, *D*, *z*, *precision*)

General hypergeometric function, where N and D are the vector of parameters in the numerator and denominator respectively, evaluated at the complex argument z.

This function implements hypergeometric functions

$${}_{p}F_{q}((a_{i})_{1 < =i < =p}, (b_{j})_{1 < =j < =q}; z) = \sum_{n > =0} (\prod_{1 < =i < =p} (a_{i})_{n}) / (\prod_{1 < =j < =q} (b_{j})_{n})(z^{n}) / (n!),$$

where $(a)_n = a(a+1)...(a+n-1)$ is the rising Pochammer symbol. For this to make sense, none of the b_j must be a negative or zero integer. The corresponding general GP command is

Whenever p = 1 or q = 1, a one-element vector can be replaced by the element it contains. Whenever p = 0 or q = 0, an empty vector can be omitted. For instance hypergeom(,b,z) computes ${}_0F_1(;b;z)$.

We distinguish three kinds of such functions according to their radius of convergence R:

- q >= p: R = oo.
- q = p 1: R = 1. Nonetheless, by integral representations, ${}_{p}F_{q}$ can be analytically continued outside the disc of convergence.
- $q \le p-2$: R = 0. By integral representations, one can make sense of the function in a suitable domain.

The list of implemented functions and their domain of validity in our implementation is as follows:

F01: hypergeom(, a, z) (or [a]). This is essentially a Bessel function and computed as such. R = oo.

F10: hypergeom(a,,z) This is $(1-z)^{-a}$.

F11: hypergeom(a,b,z) is the Kummer confluent hypergeometric function, computed by summing the series. R = oo

F20: hypergeom([a,b],,z). R = 0, computed as

$$(1)/(\Gamma(a))\int_0^o ot^{a-1}(1-zt)^{-b}e^{-t}dt.$$

F21: hypergeom([a,b],c,z) (or [c]). R = 1, extended by

$$(\Gamma(c))/(\Gamma(b)\Gamma(c-b))\int_0^1 t^{b-1}(1-t)^{c-b-1}(1-zt)^a dt.$$

This is Gauss's Hypergeometric function, and almost all of the implementation work is done for this function.

F31: hypergeom([a,b,c],d,z) (or [d]). R = 0, computed as

$$(1)/(\Gamma(a)) \int_0^o ot^{a-1} e_2^{-t} F_1(b,c;d;tz) dt.$$

F32: hypergeom([a,b,c],[d,e],z). R = 1, extended by

$$(\Gamma(e))/(\Gamma(c)\Gamma(e-c))\int_0^1 t^{c-1}(1-t)_2^{e-c-1}F_1(a,b;d;tz)dt.$$

For other inputs: if R = oo or R = 1 and $||z|| < 1 - \varepsilon$ is not too close to the circle of convergence, we simply sum the series.

This identity is due to Bercu.

hyperu(*a*, *b*, *z*, *precision*)

U-confluent hypergeometric function with complex parameters a, b, z. Note that ${}_2F_0(a, b, z) = (-z)^{-a}U(a, a + 1 - b, -1/z)$,

? hyperu(1, 3/2, I) %1 = 0.23219... - 0.80952...*I ? -I * hypergeom([1, 1+1-3/2], [], -1/I) %2 = 0.23219... - 0.80952...*I

idealadd(nf, x, y)

Sum of the two ideals x and y in the number field nf. The result is given in HNF.

```
? K = nfinit(x^2 + 1);
? a = idealadd(K, 2, x + 1) \\ ideal generated by 2 and 1+I
%2 =
[2 1]
[0 1]
? pr = idealprimedec(K, 5)[1]; \\ a prime ideal above 5
? idealadd(K, a, pr) \\ coprime, as expected
%4 =
[1 0]
[0 1]
```

This function cannot be used to add arbitrary \mathbb{Z} -modules, since it assumes that its arguments are ideals:

```
? b = Mat([1,0]~);
? idealadd(K, b, b) \\ only square t_MATs represent ideals
*** idealadd: nonsquare t_MAT in idealtyp.
? c = [2, 0; 2, 0]; idealadd(K, c, c) \\ nonsense
%6 =
[2 0]
[0 2]
? d = [1, 0; 0, 2]; idealadd(K, d, d) \\ nonsense
%7 =
[1 0]
[0 1]
```

In the last two examples, we get wrong results since the matrices c and d do not correspond to an ideal: the \mathbb{Z} -span of their columns (as usual interpreted as coordinates with respect to the integer basis K.zk) is not an O_K -module. To add arbitrary \mathbb{Z} -modules generated by the columns of matrices A and B, use mathnf(concat(A,B)).

idealaddtoone(nf, x, y)

x and y being two co-prime integral ideals (given in any form), this gives a two-component row vector [a, b] such that $a \in x, b \in y$ and a + b = 1.

The alternative syntax idealaddtoone(nf, v), is supported, where v is a k-component vector of ideals (given in any form) which sum to \mathbb{Z}_K . This outputs a k-component vector e such that $e[i] \in x[i]$ for $1 \le i \le k$ and $\sum_{1 \le i \le k} e[i] = 1$.

idealappr(nf, x, flag)

If x is a fractional ideal (given in any form), gives an element α in nf such that for all prime ideals p such that the valuation of x at p is nonzero, we have $v_p(\alpha) = v_p(x)$, and $v_p(\alpha) \ge 0$ for all other p.

The argument x may also be given as a prime ideal factorization, as output by idealfactor, but allowing zero exponents. This yields an element α such that for all prime ideals p occurring in x, $v_p(\alpha) = v_p(x)$; for all other prime ideals, $v_p(\alpha) \ge 0$.

flag is deprecated (ignored), kept for backward compatibility.

idealchinese(nf, x, y)

x being a prime ideal factorization (i.e. a 2-columns matrix whose first column contains prime ideals and the second column contains integral exponents), y a vector of elements in nf indexed by the ideals in x, computes an element b such that

 $v_p(b-y_p) \ge v_p(x)$ for all prime ideals in x and $v_p(b) \ge 0$ for all other p.

```
? K = nfinit(t<sup>2</sup>-2);
? x = idealfactor(K, 2<sup>2</sup>*3)
%2 =
[[2, [0, 1]~, 2, 1, [0, 2; 1, 0]] 4]
[ [3, [3, 0]~, 1, 2, 1] 1]
? y = [t,1];
? idealchinese(K, x, y)
%4 = [4, -3]~
```

The argument x may also be of the form [x, s] where the first component is as above and s is a vector of signs, with r_1 components s_i in -1, 0, 1: if σ_i denotes the *i*-th real embedding of the number field, the element b returned satisfies further $sign(\sigma_i(b)) = s_i$ for all *i* such that $s_i = 1$. In other words, the sign is fixed to s_i at the *i*-th embedding whenever s_i is nonzero.

```
? idealchinese(K, [x, [1,1]], y)
%5 = [16, -3]~
? idealchinese(K, [x, [-1,-1]], y)
%6 = [-20, -3]~
? idealchinese(K, [x, [1,-1]], y)
%7 = [4, -3]~
```

If y is omitted, return a data structure which can be used in place of x in later calls and allows to solve many chinese remainder problems for a given x more efficiently.

? C = idealchinese(K, [x, [1,1]]); ? idealchinese(K, C, y) \\ as above %9 = [16, -3]~

```
? for(i=1,10^4, idealchinese(K,C,y)) \\ ... but faster !
time = 80 ms.
? for(i=1,10^4, idealchinese(K,[x,[1,1]],y))
time = 224 ms.
```

Finally, this structure is itself allowed in place of x, the new s overriding the one already present in the structure. This allows to initialize for different sign conditions more efficiently when the underlying ideal factorization remains the same.

```
? D = idealchinese(K, [C, [1,-1]]); \\ replaces [1,1]
? idealchinese(K, D, y)
%13 = [4, -3]~
? for(i=1,10^4,idealchinese(K,[C,[1,-1]]))
time = 40 ms. \\ faster than starting from scratch
? for(i=1,10^4,idealchinese(K,[x,[1,-1]]))
time = 128 ms.
```

idealcoprime(nf, x, y)

Given two integral ideals x and y in the number field nf, returns a β in the field, such that β .x is an integral ideal coprime to y.

idealdiv(nf, x, y, flag)

Quotient $x.y^{-1}$ of the two ideals x and y in the number field nf. The result is given in HNF.

If *flag* is nonzero, the quotient $x.y^{-1}$ is assumed to be an integral ideal. This can be much faster when the norm of the quotient is small even though the norms of x and y are large. More precisely, the algorithm cheaply removes all maximal ideals above rational primes such that $v_p(Nx) = v_p(Ny)$.

idealdown(nf, x)

Let nf be a number field as output by nfinit, and x a fractional ideal. This function returns the nonnegative rational generator of $x \cap \mathbb{Q}$. If x is an extended ideal, the extended part is ignored.

```
? nf = nfinit(y<sup>2</sup>+1);
? idealdown(nf, -1/2)
%2 = 1/2
? idealdown(nf, (y+1)/3)
%3 = 2/3
? idealdown(nf, [2, 11]~)
%4 = 125
? x = idealprimedec(nf, 2)[1]; idealdown(nf, x)
%5 = 2
? idealdown(nf, [130, 94; 0, 2])
%6 = 130
```

idealfactor(nf, x, lim)

Factors into prime ideal powers the ideal x in the number field nf. The output format is similar to the factor function, and the prime ideals are represented in the form output by the idealprimedec function. If *lim* is set, return partial factorization, including only prime ideals above rational primes < lim.

```
? nf = nfinit(x^3-2);
? idealfactor(nf, x) \\ a prime ideal above 2
%2 =
[[2, [0, 1, 0]~, 3, 1, ...] 1]
```

```
? A = idealhnf(nf, 6*x, 4+2*x+x^2)
%3 =
[6 0 4]
[0 6 2]
[0 0 1]
? idealfactor(nf, A)
%4 =
 [[2, [0, 1, 0]~, 3, 1, ...] 2]
[[3, [1, 1, 0]~, 3, 1, ...] 2]
? idealfactor(nf, A, 3) \\ restrict to primes above p < 3
%5 =
 [[2, [0, 1, 0]~, 3, 1, ...] 2]
```

idealfactorback(nf, f, e, flag)

Gives back the ideal corresponding to a factorization. The integer 1 corresponds to the empty factorization. If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization, as produced by idealfactor.

```
? nf = nfinit(y<sup>2</sup>+1); idealfactor(nf, 4 + 2*y)
%1 =
[[2, [1, 1]~, 2, 1, [1, 1]~] 2]
[[5, [2, 1]~, 1, 1, [-2, 1]~] 1]
? idealfactorback(nf, %)
%2 =
[10 4]
[0 2]
? f = %1[,1]; e = %1[,2]; idealfactorback(nf, f, e)
%3 =
[10 4]
[0 2]
? % == idealhnf(nf, 4 + 2*y)
%4 = 1
```

If flag is nonzero, perform ideal reductions (idealred) along the way. This is most useful if the ideals involved are all *extended* ideals (for instance with trivial principal part), so that the principal parts extracted by idealred are not lost. Here is an example:

```
? f = vector(#f, i, [f[i], [;]]); \\ transform to extended ideals
? idealfactorback(nf, f, e, 1)
%6 = [[1, 0; 0, 1], [2, 1; [2, 1]~, 1]]
? nffactorback(nf, %[2])
%7 = [4, 2]~
```

The extended ideal returned in %6 is the trivial ideal 1, extended with a principal generator given in factored form. We use nffactorback to recover it in standard form.

idealfrobenius(nf, gal, pr)

Let K be the number field defined by nf and assume K/\mathbb{Q} be a Galois extension with Galois group given gal = galoisinit(nf), and that pr is an unramified prime ideal p in prid format. This function returns a permutation of gal.group which defines the Frobenius element Frob_p attached to p. If p is the unique prime number in p, then $\operatorname{Frob}(x) = x^p \mod p$ for all $x \in \mathbb{Z}_K$.

```
? nf = nfinit(polcyclo(31));
? gal = galoisinit(nf);
? pr = idealprimedec(nf,101)[1];
? g = idealfrobenius(nf,gal,pr);
? galoispermtopol(gal,g)
%5 = x^8
```

This is correct since 101 = 8mod31.

idealhnf(nf, u, v)

Gives the Hermite normal form of the ideal $u\mathbb{Z}_K + v\mathbb{Z}_K$, where u and v are elements of the number field K defined by *nf*.

```
? nf = nfinit(y<sup>3</sup> - 2);
? idealhnf(nf, 2, y+1)
%2 =
[1 0 0]
[0 1 0]
[0 0 1]
? idealhnf(nf, y/2, [0,0,1/3]~)
%3 =
[1/3 0 0]
[0 1/6 0]
[0 0 1/6]
```

If *b* is omitted, returns the HNF of the ideal defined by *u*: *u* may be an algebraic number (defining a principal ideal), a maximal ideal (as given by idealprimedec or idealfactor), or a matrix whose columns give generators for the ideal. This last format is a little complicated, but useful to reduce general modules to the canonical form once in a while:

- if strictly less than $N = [K : \mathbb{Q}]$ generators are given, u is the \mathbb{Z}_K -module they generate,
- if N or more are given, it is assumed that they form a Z-basis of the ideal, in particular that the matrix has
 maximal rank N. This acts as mathnf since the Z_K-module structure is (taken for granted hence) not taken
 into account in this case.

? idealhnf(nf, idealprimedec(nf,2)[1])
%4 =
[2 0 0]
[0 1 0]
[0 0 1]
? idealhnf(nf, [1,2;2,3;3,4])
%5 =
[1 0 0]
[0 1 0]
[0 0 1]

Finally, when K is quadratic with discriminant D_K , we allow u = Qfb(a,b,c), provided $b^2 - 4ac = D_K$. As usual, this represents the ideal $a\mathbb{Z} + (1/2)(-b + \sqrt{D_K})\mathbb{Z}$.

idealintersect(nf, A, B)

Intersection of the two ideals A and B in the number field nf. The result is given in HNF.

? nf = nfinit(x^2+1); ? idealintersect(nf, 2, x+1) %2 = [2 0] [0 2]

This function does not apply to general \mathbb{Z} -modules, e.g. orders, since its arguments are replaced by the ideals they generate. The following script intersects \mathbb{Z} -modules A and B given by matrices of compatible dimensions with integer coefficients:

```
ZM_intersect(A,B) =
{ my(Ker = matkerint(concat(A,B)));
 mathnf( A * Ker[1..#A,] )
}
```

idealinv(nf, x)

Inverse of the ideal x in the number field nf, given in HNF. If x is an extended ideal, its principal part is suitably updated: i.e. inverting [I, t], yields $[I^{-1}, 1/t]$.

idealismaximal(nf, x)

Given nf a number field as output by nfinit and an ideal x, return 0 if x is not a maximal ideal. Otherwise return a prid structure nf attached to the ideal. This function uses ispseudoprime and may return a wrong result in case the underlying rational pseudoprime is not an actual prime number: apply isprime(pr.p) to guarantee correctness. If x is an extended ideal, the extended part is ignored.

```
? K = nfinit(y<sup>2</sup> + 1);
? idealismaximal(K, 3) \\ 3 is inert
%2 = [3, [3, 0]~, 1, 2, 1]
? idealismaximal(K, 5) \\ 5 is not
%3 = 0
? pr = idealprimedec(K,5)[1] \\ already a prid
%4 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
? idealismaximal(K, pr) \\ trivial check
%5 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
? x = idealhnf(K, pr)
%6 =
[5 3]
[0 1]
? idealismaximal(K, x) \\ converts from matrix form to prid
%7 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
```

This function is noticeably faster than idealfactor since it never involves an actually factorization, in particular when $x \cap \mathbb{Z}$ is not a prime number.

idealispower(nf, A, n, B)

Let *nf* be a number field and n > 0 be a positive integer. Return 1 if the fractional ideal $A = B^n$ is an *n*-th power and 0 otherwise. If the argument B is present, set it to the *n*-th root of A, in HNF.

```
? K = nfinit(x<sup>3</sup> - 2);
? A = [46875, 30966, 9573; 0, 3, 0; 0, 0, 3];
? idealispower(K, A, 3, &B)
%3 = 1
? B
%4 =
[75 22 41]
[ 0 1 0]
[ 0 0 1]
? A = [9375, 2841, 198; 0, 3, 0; 0, 0, 3];
? idealispower(K, A, 3)
%5 = 0
```

ideallist(nf, bound, flag)

Computes the list of all ideals of norm less or equal to *bound* in the number field *nf*. The result is a row vector with exactly *bound* components. Each component is itself a row vector containing the information about ideals of a given norm, in no specific order, depending on the value of flag:

The possible values of flag are:

0: give the *bid* attached to the ideals, without generators.

1: as 0, but include the generators in the *bid*.

2: in this case, *nf* must be a *bnf* with units. Each component is of the form [bid, U], where *bid* is as case 0 and U is a vector of discrete logarithms of the units. More precisely, it gives the ideallog s with respect to *bid* of $(\zeta, u_1, ..., u_r)$ where ζ is the torsion unit generator bnf.tu[2] and (u_i) are the fundamental units in bnf.fu. This structure is technical, and only meant to be used in conjunction with bnrclassnolist or bnrdisclist.

3: as 2, but include the generators in the *bid*.

4: give only the HNF of the ideal.

```
? nf = nfinit(x^2+1);
? L = ideallist(nf, 100);
? L[1]
%3 = [[1, 0; 0, 1]] \\ A single ideal of norm 1
? #L[65]
%4 = 4 \\ There are 4 ideals of norm 4 in Z[i]
```

If one wants more information, one could do instead:

```
? nf = nfinit(x^2+1);
? L = ideallist(nf, 100, 0);
? l = L[25]; vector(#l, i, 1[i].clgp)
%3 = [[20, [20]], [16, [4, 4]], [20, [20]]]
? 1[1].mod
%4 = [[25, 18; 0, 1], []]
? 1[2].mod
%5 = [[5, 0; 0, 5], []]
? 1[3].mod
%6 = [[25, 7; 0, 1], []]
```

where we ask for the structures of the $(\mathbb{Z}[i]/I)^*$ for all three ideals of norm 25. In fact, for all moduli with finite part of norm 25 and trivial Archimedean part, as the last 3 commands show. See ideallistarch to treat general moduli.

ideallistarch(nf, list, arch)

list is a vector of vectors of bid's, as output by ideallist with flag 0 to 3. Return a vector of vectors with the same number of components as the original *list*. The leaves give information about moduli whose finite part is as in original list, in the same order, and Archimedean part is now *arch* (it was originally trivial). The information contained is of the same kind as was present in the input; see ideallist, in particular the meaning of *flag*.

```
? bnf = bnfinit(x^2-2);
? bnf.sign
%2 = [2, 0] \\ two places at infinity
? L = ideallist(bnf, 100, 0);
? l = L[98]; vector(#1, i, 1[i].clgp)
%4 = [[42, [42]], [36, [6, 6]], [42, [42]]]
? La = ideallistarch(bnf, L, [1,1]); \\ add them to the modulus
? l = La[98]; vector(#1, i, 1[i].clgp)
%6 = [[168, [42, 2, 2]], [144, [6, 6, 2, 2]], [168, [42, 2, 2]]]
```

Of course, the results above are obvious: adding t places at infinity will add t copies of $\mathbb{Z}/2\mathbb{Z}$ to $(\mathbb{Z}_K/f)^*$. The following application is more typical:

```
? L = ideallist(bnf, 100, 2); \\ units are required now
? La = ideallistarch(bnf, L, [1,1]);
? H = bnrclassnolist(bnf, La);
```

? H[98];		
%4 = [2, 12, 2]		

ideallog(nf, x, bid)

nf is a number field, *bid* is as output by idealstar(nf, D,...) and x an element of nf which must have valuation equal to 0 at all prime ideals in the support of D and need not be integral. This function computes the discrete logarithm of x on the generators given in :emphasis:`bid.gen`. In other words, if g_i are these generators, of orders d_i respectively, the result is a column vector of integers (x_i) such that $0 \le x_i < d_i$ and

$$x = \prod_{i} g_i^{x_i}(mod^*D).$$

Note that when the support of D contains places at infinity, this congruence implies also sign conditions on the attached real embeddings. See znlog for the limitations of the underlying discrete log algorithms.

When nf is omitted, take it to be the rational number field. In that case, x must be a t_INT and *bid* must have been initialized by znstar(N, 1).

idealmin(nf, ix, vdir)

This function is useless and kept for backward compatibility only, use :literal: `idealred`. Computes a pseudominimum of the ideal x in the direction vdir in the number field nf.

idealmul(nf, x, y, flag)

Ideal multiplication of the ideals x and y in the number field nf; the result is the ideal product in HNF. If either x or y are extended ideals, their principal part is suitably updated: i.e. multiplying [I, t], [J, u] yields [IJ, tu]; multiplying I and [J, u] yields [IJ, u].

```
? nf = nfinit(x^2 + 1);
? idealmul(nf, 2, x+1)
%2 =
[4 2]
[0 2]
? idealmul(nf, [2, x], x+1) \\ extended ideal * ideal
%3 = [[4, 2; 0, 2], x]
? idealmul(nf, [2, x], [x+1, x]) \\ two extended ideals
%4 = [[4, 2; 0, 2], [-1, 0]~]
```

If *flag* is nonzero, reduce the result using idealred.

idealnorm(nf, x)

Computes the norm of the ideal x in the number field nf.

idealnumden(nf, x)

Returns [A, B], where A, B are coprime integer ideals such that x = A/B, in the number field nf.

? nf = nfinit(x^2+1); ? idealnumden(nf, (x+1)/2) %2 = [[1, 0; 0, 1], [2, 1; 0, 1]]

idealpow(nf, x, k, flag)

Computes the k-th power of the ideal x in the number field nf; $k \in \mathbb{Z}$. If x is an extended ideal, its principal part is suitably updated: i.e. raising [I, t] to the k-th power, yields $[I^k, t^k]$.

If flag is nonzero, reduce the result using idealred, throughout the (binary) powering process; in particular, this is not the same as idealpow(nf, x, k) followed by reduction.

idealprimedec(nf, p, f)

Computes the prime ideal decomposition of the (positive) prime number p in the number field K represented by *nf*. If a nonprime p is given the result is undefined. If f is present and nonzero, restrict the result to primes of residue degree $\leq f$.

The result is a vector of *prid* structures, each representing one of the prime ideals above p in the number field nf. The representation pr = [p, a, e, f, mb] of a prime ideal means the following: a is an algebraic integer in the maximal order \mathbb{Z}_K and the prime ideal is equal to $p = p\mathbb{Z}_K + a\mathbb{Z}_K$; e is the ramification index; f is the residual index; finally, mb is the multiplication table attached to the algebraic integer b is such that $p^{-1} = \mathbb{Z}_K + b/p\mathbb{Z}_K$, which is used internally to compute valuations. In other words if p is inert, then mb is the integer 1, and otherwise it is a square t_MAT whose j-th column is b.nf.zk[j].

The algebraic number a is guaranteed to have a valuation equal to 1 at the prime ideal (this is automatic if e > 1).

The components of pr should be accessed by member functions: pr.p, pr.e, pr.f, and pr.gen (returns the vector [p, a]):

```
? K = nfinit(x^3-2);
? P = idealprimedec(K, 5);
? #P \\ 2 primes above 5 in Q(2^(1/3))
%3 = 2
? [p1,p2] = P;
? [p1.e, p1.f] \\ the first is unramified of degree 1
%5 = [1, 1]
? [p2.e, p2.f] \\ the second is unramified of degree 2
%6 = [1, 2]
? p1.gen
%7 = [5, [2, 1, 0]~]
? nfbasistoalg(K, %[2]) \\ a uniformizer for p1
%8 = Mod(x + 2, x^3 - 2)
? #idealprimedec(K, 5, 1) \\ restrict to f = 1
%9 = 1 \\ now only p1
```

idealprincipalunits(nf, pr, k)

Given a prime ideal in idealprimedec format, returns the multiplicative group $(1+pr)/(1+pr^k)$ as an abelian group. This function is much faster than idealstar when the norm of pr is large, since it avoids (useless) work in the multiplicative group of the residue field.

```
? K = nfinit(y^2+1);
? P = idealprimedec(K,2)[1];
? G = idealprincipalunits(K, P, 20);
? G.cyc
%4 = [512, 256, 4] \\ Z/512 x Z/256 x Z/4
? G.gen
%5 = [[-1, -2]~, 1021, [0, -1]~] \\ minimal generators of given order
```

idealramgroups(nf, gal, pr)

Let K be the number field defined by nf and assume that K/\mathbb{Q} is Galois with Galois group G given by gal = galoisinit(nf). Let pr be the prime ideal P in prid format. This function returns a vector g of subgroups of gal as follows:

- g[1] is the decomposition group of P,
- g[2] is $G_0(P)$, the inertia group of P,

and for $i \ge 2$,

• g[i] is $G_{i-2}(P)$, the i-2-th ramification group of P.

The length of g is the number of nontrivial groups in the sequence, thus is 0 if e = 1 and f = 1, and 1 if f > 1 and e = 1. The following function computes the cardinality of a subgroup of G, as given by the components of g:

card(H) =my(o=H[2]); prod(i=1,#o,o[i]);

```
? nf=nfinit(x^6+3); gal=galoisinit(nf); pr=idealprimedec(nf,3)[1];
? g = idealramgroups(nf, gal, pr);
? apply(card,g)
%3 = [6, 6, 3, 3, 3] \\ cardinalities of the G_i
```

```
? nf=nfinit(x^6+108); gal=galoisinit(nf); pr=idealprimedec(nf,2)[1];
? iso=idealramgroups(nf,gal,pr)[2]
%5 = [[Vecsmall([2, 3, 1, 5, 6, 4])], Vecsmall([3])]
? nfdisc(galoisfixedfield(gal,iso,1))
%6 = -3
```

The field fixed by the inertia group of 2 is not ramified at 2.

idealred(nf, I, v)

LLL reduction of the ideal I in the number field K attached to nf, along the direction v. The v parameter is best left omitted, but if it is present, it must be an nf.r1 + nf.r2-component vector of *nonnegative* integers. (What counts is the relative magnitude of the entries: if all entries are equal, the effect is the same as if the vector had been omitted.)

This function finds an $a \in K^*$ such that J = (a)I is "small" and integral (see the end for technical details). The result is the Hermite normal form of the "reduced" ideal J.

```
? K = nfinit(y<sup>2</sup>+1);
? P = idealprimedec(K,5)[1];
? idealred(K, P)
%3 =
[1 0]
[0 1]
```

More often than not, a principal ideal yields the unit ideal as above. This is a quick and dirty way to check if ideals are principal, but it is not a necessary condition: a nontrivial result does not prove that the ideal is nonprincipal. For guaranteed results, see **bnfisprincipal**, which requires the computation of a full **bnf** structure.

If the input is an extended ideal [I, s], the output is [J, sa]; in this way, one keeps track of the principal ideal part:

```
? idealred(K, [P, 1])
%5 = [[1, 0; 0, 1], [2, -1]~]
```

meaning that P is generated by [2, -1]. The number field element in the extended part is an algebraic number in any form *or* a factorization matrix (in terms of number field elements, not ideals!). In the latter case, elements stay in factored form, which is a convenient way to avoid coefficient explosion; see also idealpow.

Technical note. The routine computes an LLL-reduced basis for the lattice I^{-1} equipped with the quadratic form

$$||||x||||_{v}^{2} = \sum_{i=1}^{r_{1}+r_{2}} 2^{v_{i}} \varepsilon_{i} ||\sigma_{i}(x)||^{2},$$

where as usual the σ_i are the (real and) complex embeddings and $\varepsilon_i = 1$, resp. 2, for a real, resp. complex place. The element *a* is simply the first vector in the LLL basis. The only reason you may want to try to change some directions and set some $v_i! = 0$ is to randomize the elements found for a fixed ideal, which is heuristically useful in index calculus algorithms like bnfinit and bnfisprincipal.

Even more technical note. In fact, the above is a white lie. We do not use $|||.|||_v$ exactly but a rescaled rounded variant which gets us faster and simpler LLLs. There's no harm since we are not using any theoretical property of *a* after all, except that it belongs to I^{-1} and that aI is "expected to be small".

idealredmodpower(nf, x, n, B)

Let *nf* be a number field, x an ideal in *nf* and n > 0 be a positive integer. Return a number field element b such that $xb^n = v$ is small. If x is integral, then v is also integral.

More precisely, idealnumden reduces the problem to x integral. Then, factoring out the prime ideals dividing a rational prime $p \le B$, we rewrite $x = IJ^n$ where the ideals I and J are both integral and I is B-smooth. Then we return a small element b in J^{-1} .

The bound B avoids a costly complete factorization of x; as soon as the *n*-core of x is B-smooth (i.e., as soon as I is *n*-power free), then J is as large as possible and so is the expected reduction.

```
? T = x^6+108; nf = nfinit(T); a = Mod(x,T);
? setrand(1); u = (2*a^2+a+3)*random(2^1000*x^6)^6;
? sizebyte(u)
%3 = 4864
? b = idealredmodpower(nf,u,2);
? v2 = nfeltmul(nf,u, nfeltpow(nf,b,2))
%5 = [34, 47, 15, 35, 9, 3]~
? b = idealredmodpower(nf,u,6);
? v6 = nfeltmul(nf,u, nfeltpow(nf,b,6))
%7 = [3, 0, 2, 6, -7, 1]~
```

The last element v6, obtained by reducing modulo 6-th powers instead of squares, looks smaller than v2 but its norm is actually a little larger:

```
? idealnorm(nf,v2)
%8 = 81309
? idealnorm(nf,v6)
%9 = 731781
```

idealstar(nf, N, flag, cycmod)

Outputs a bid structure, necessary for computing in the finite abelian group $G = (\mathbb{Z}_K/N)^*$. Here, *nf* is a number field and *N* is a *modulus*: either an ideal in any form, or a row vector whose first component is an ideal and whose second component is a row vector of r_1 0 or 1. Ideals can also be given by a factorization into prime ideals, as produced by idealfactor.

If the positive integer cycmod is present, only compute the group modulo cycmod-th powers, which may save a lot of time when some maximal ideals in the modulus have a huge residue field. Whereas you might only be interested in quadratic or cubic residuosity; see also bnrinit for applications in class field theory.

This *bid* is used in ideallog to compute discrete logarithms. It also contains useful information which can be conveniently retrieved as :emphasis:`bid.mod` (the modulus), :emphasis:`bid.clgp` (G as a finite abelian group), :emphasis:`bid.no` (the cardinality of G), :emphasis:`bid.cyc` (elementary divisors) and :emphasis:`bid.gen` (generators).

If flag = 1 (default), the result is a bid structure without generators: they are well defined but not explicitly computed, which saves time.

If flag = 2, as flag = 1, but including generators.

If flag = 0, only outputs $(\mathbb{Z}_K/N)^*$ as an abelian group, i.e as a 3-component vector [h, d, g]: h is the order, d is the vector of SNF cyclic components and g the corresponding generators.

If *nf* is omitted, we take it to be the rational number fields, N must be an integer and we return the structure of $(\mathbb{Z}/N\mathbb{Z})^*$. In other words idealstar(, N, flag) is short for

```
idealstar(nfinit(x), N, flag)
```

but faster. The alternative syntax znstar(N, flag) is also available for an analogous effect but, due to an unfortunate historical oversight, the default value of flag is different in the two functions (znstar does not initialize by default, you probably want znstar(N, 1)).

idealtwoelt(nf, x, a)

Computes a two-element representation of the ideal x in the number field nf, combining a random search and an approximation theorem; x is an ideal in any form (possibly an extended ideal, whose principal part is ignored)

• When called as idealtwoelt(nf,x), the result is a row vector $[a, \alpha]$ with two components such that $x = a\mathbb{Z}_K + \alpha\mathbb{Z}_K$ and a is chosen to be the positive generator of $x \cap \mathbb{Z}$, unless x was given as a principal ideal in which case we may choose a = 0. The algorithm uses a fast lazy factorization of $x \cap \mathbb{Z}$ and runs in randomized polynomial time.

```
? K = nfinit(t^5-23);
? x = idealhnf(K, t^2*(t+1), t^3*(t+1))
%2 = \\ some random ideal of norm 552*23
[552 23 23 529 23]
[ 0 23 0 0 0]
[ 0 0 1 0 0]
[ 0 0 0 1 0]
[ 0 0 0 1 0]
[ 0 0 0 1 0]
[ 0 0 0 1]
? [a,alpha] = idealtwoelt(K, x)
%3 = [552, [23, 0, 1, 0, 0]~]
? nfbasistoalg(K, alpha)
%4 = Mod(t^2 + 23, t^5 - 23)
```

• When called as idealtwoelt(nf, x, a) with an explicit nonzero a supplied as third argument, the function assumes that $a \in x$ and returns $\alpha \in x$ such that $x = a\mathbb{Z}_K + \alpha\mathbb{Z}_K$. Note that we must factor a in this case, and the algorithm is generally slower than the default variant and gives larger generators:

? alpha2 = idealtwoelt(K, x, 552) %5 = [-161, -161, -183, -207, 0]~ ? idealhnf(K, 552, alpha2) == x %6 = 1

Note that, in both cases, the return value is *not* recognized as an ideal by GP functions; one must use idealhnf as above to recover a valid ideal structure from the two-element representation.

idealval(nf, x, pr)

Gives the valuation of the ideal x at the prime ideal pr in the number field nf, where pr is in idealprimedec format. The valuation of the 0 ideal is +00.

imag(x)

Imaginary part of x. When x is a quadratic number, this is the coefficient of ω in the "canonical" integral basis $(1, \omega)$.

? imag(3 + I)
%1 = 1
? x = 3 + quadgen(-23);
? imag(x) \\ as a quadratic number
%3 = 1
? imag(x * 1.) \\ as a complex number
%4 = 2.3979157616563597707987190320813469600

incgam(*s*, *x*, *g*, *precision*)

Incomplete gamma function $\int_x^o oe^{-t}t^{s-1}dt$, extended by analytic continuation to all complex x, s not both 0. The relative error is bounded in terms of the precision of s (the accuracy of x is ignored when determining the output precision). When g is given, assume that $g = \Gamma(s)$. For small ||x||, this will speed up the computation.

incgamc(*s*, *x*, *precision*)

Complementary incomplete gamma function. The arguments x and s are complex numbers such that s is not a pole of Γ and ||x||/(||s|| + 1) is not much larger than 1 (otherwise the convergence is very slow). The result returned is $\int_0^x e^{-t} t^{s-1} dt$.

input()

Reads a string, interpreted as a GP expression, from the input file, usually standard input (i.e. the keyboard). If a sequence of expressions is given, the result is the result of the last expression of the sequence. When using this instruction, it is useful to prompt for the string by using the print1 function. Note that in the present version 2.19 of pari.el, when using gp under GNU Emacs (see emacs (in the PARI manual)) one *must* prompt for the string, with a string which ends with the same prompt as any of the previous ones (a "? " will do for instance).

install(name, code, gpname, lib)

Loads from dynamic library *lib* the function *name*. Assigns to it the name *gpname* in this gp session, with *prototype code* (see below). If *gpname* is omitted, uses *name*. If *lib* is omitted, all symbols known to gp are available: this includes the whole of libpari.so and possibly others (such as libc.so).

Most importantly, install gives you access to all nonstatic functions defined in the PARI library. For instance, the function

GEN addii(GEN x, GEN y)

adds two PARI integers, and is not directly accessible under gp (it is eventually called by the + operator of course):

```
? install("addii", "GG")
? addii(1, 2)
%1 = 3
```

It also allows to add external functions to the gp interpreter. For instance, it makes the function system obsolete:

```
? install(system, vs, sys,/*omitted*/)
? sys("ls gp*")
gp.c gp.h gp_rl.c
```

This works because system is part of libc.so, which is linked to gp. It is also possible to compile a shared library yourself and provide it to gp in this way: use gp2c, or do it manually (see the modules_build variable in pari.cfg for hints).

Re-installing a function will print a warning and update the prototype code if needed. However, it will not reload a symbol from the library, even if the latter has been recompiled.

Prototype. We only give a simplified description here, covering most functions, but there are many more possibilities. The full documentation is available in libpari.dvi, see

??prototype

- First character i, l, u, v : return type int / long / ulong / void. (Default: GEN)
- One letter for each mandatory argument, in the same order as they appear in the argument list: G (GEN), & (GEN*), L (long), U (ulong), s (char *), n (variable).
- p to supply realprecision (usually long prec in the argument list), b to supply realbitprecision (usually long bitprec), P to supply seriesprecision (usually long precdl).

We also have special constructs for optional arguments and default values:

- DG (optional GEN, NULL if omitted),
- D& (optional GEN*, NULL if omitted),
- Dn (optional variable, -1 if omitted),

For instance the prototype corresponding to

```
long issquareall(GEN x, GEN *n = NULL)
```

is 1GD&.

Caution. This function may not work on all systems, especially when gp has been compiled statically. In that case, the first use of an installed function will provoke a Segmentation Fault (this should never happen with a dynamically linked executable). If you intend to use this function, please check first on some harmless example such as the one above that it works properly on your machine.

intformal(x, v)

formal integration of x with respect to the variable v (wrt. the main variable if v is omitted). Since PARI cannot represent logarithmic or arctangent terms, any such term in the result will yield an error:

```
? intformal(x^2)
%1 = 1/3*x^3
? intformal(x^2, y)
%2 = y*x^2
? intformal(1/x)
*** at top-level: intformal(1/x)
*** ^------
*** intformal: domain error in intformal: residue(series, pole) != 0
```

The argument x can be of any type. When x is a rational function, we assume that the base ring is an integral domain of characteristic zero.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the integral is taken with respect to the main variable used in the base ring R. In particular, it is meaningless to integrate with respect to the main variable of x.mod:

```
? intformal(Mod(1,x^2+1), 'x)
*** intformal: incorrect priority in intformal: variable x = x
```

intnumgaussinit(n, precision)

Initialize tables for *n*-point Gauss-Legendre integration of a smooth function f on a compact interval [a, b]. If n is omitted, make a default choice n B/4, where B is realbit precision, suitable for analytic functions on [-1, 1]. The error is bounded by

$$((b-a)^{2n+1}(n!)^4)/((2n+1)!(2n)!)(f^{(2n)})/((2n)!)(\xi), a < \xi < b.$$

If r denotes the distance of the nearest pole to the interval [a, b], then this is of the order of $((b-a)/(4r))^{2n}$. In particular, the integral must be subdivided if the interval length b-a becomes close to 4r. The default choice n B/4 makes this quantity of order 2^{-B} when b-a=r, as is the case when integrating 1/(1+t) on [0,1] for instance. If the interval length increases, n should be increased as well.

Specifically, the function returns a pair of vectors [x, w], where x contains the nonnegative roots of the n-th Legendre polynomial P_n and w the corresponding Gaussian integration weights $Q_n(x_j)/P'_n(x_j) = 2/((1 - x_j^2)P'_n(x_j))^2$ such that

$$\int_{-1}^{1} f(t)dt w_j f(x_j).$$

```
? T = intnumgaussinit();
? intnumgauss(t=-1,1,exp(t), T) - exp(1)+exp(-1)
%1 = -5.877471754111437540 E-39
? intnumgauss(t=-10,10,exp(t), T) - exp(10)+exp(-10)
%2 = -8.358367809712546836 E-35
? intnumgauss(t=-1,1,1/(1+t^2), T) - Pi/2 \\ b - a = 2r
%3 = -9.490148553624725335 E-22 \\ ... loses half the accuracy
? T = intnumgaussinit(50);
? intnumgauss(t=-1,1,1/(1+t^2), T) - Pi/2
%5 = -1.1754943508222875080 E-38
? intnumgauss(t=-5,5,1/(1+t^2), T) - 2*atan(5)
%6 = -1.2[...]E-8
```

On the other hand, we recommend to split the integral and change variables rather than increasing n too much, see intnumgauss.

intnuminit(a, b, m, precision)

Initialize tables for integration from *a* to *b*, where *a* and *b* are coded as in intrum. Only the compactness, the possible existence of singularities, the speed of decrease or the oscillations at infinity are taken into account, and not the values. For instance intruminit((-1, -1)) is equivalent to intruminit((0, -1/2), oo) is equivalent to intruminit((-1, -1/2), -oo); on the other hand, the order matters and intruminit((0, -1/2), (1, -1/3)) is *not* equivalent to intruminit((0, -1/3), (1, -1/2))!

If m is present, it must be nonnegative and we multiply the default number of sampling points by 2^m (increasing the running time by a similar factor).

The result is technical and liable to change in the future, but we document it here for completeness. Let $x = \phi(t)$, $t \in] -oo, oo[$ be an internally chosen change of variable, achieving double exponential decrease of the integrand at infinity. The integrator intrum will compute

$$h\sum_{\|n\| < N} \phi'(nh) F(\phi(nh))$$

for some integration step h and truncation parameter N. In basic use, let

[h, x0, w0, xp, wp, xm, wm] = intnuminit(a,b);

- *h* is the integration step
- $x_0 = \phi(0)$ and $w_0 = \phi'(0)$,
- *xp* contains the $\phi(nh)$, 0 < n < N,
- xm contains the $\phi(nh)$, 0 < -n < N, or is empty.
- wp contains the $\phi'(nh)$, 0 < n < N,
- wm contains the $\phi'(nh)$, 0 < -n < N, or is empty.

The arrays xm and wm are left empty when ϕ is an odd function. In complicated situations, intruminit may return up to 3 such arrays, corresponding to a splitting of up to 3 integrals of basic type.

If the functions to be integrated later are of the form F = f(t)k(t, z) for some kernel k (e.g. Fourier, Laplace, Mellin,...), it is useful to also precompute the values of $f(\phi(nh))$, which is accomplished by intfuncinit. The hard part is to determine the behavior of F at endpoints, depending on z.

isfundamental(D)

True (1) if D is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise. D can be input in factored form as for arithmetic functions:

```
? isfundamental(factor(-8))
%1 = 1
\\ count fundamental discriminants up to 10^8
? c = 0; forfactored(d = 1, 10^8, if (isfundamental(d), c++)); c
time = 40,840 ms.
%2 = 30396325
? c = 0; for(d = 1, 10^8, if (isfundamental(d), c++)); c
time = 1min, 33,593 ms. \\ slower !
%3 = 30396325
```

ispolygonal(x, s, N)

True (1) if the integer x is an s-gonal number, false (0) if not. The parameter s > 2 must be a t_INT. If N is given, set it to n if x is the n-th s-gonal number.

```
? ispolygonal(36, 3, &N)
%1 = 1
? N
```

ispower(x, k, n)

If k is given, returns true (1) if x is a k-th power, false (0) if not. What it means to be a k-th power depends on the type of x; see issquare for details.

If k is omitted, only integers and fractions are allowed for x and the function returns the maximal $k \ge 2$ such that $x = n^k$ is a perfect power, or 0 if no such k exist; in particular ispower(-1), ispower(0), and ispower(1) all return 0.

If a third argument n is given and x is indeed a k-th power, sets n to a k-th root of x.

For a t_FFELT x, instead of omitting k (which is not allowed for this type), it may be natural to set

 $| k = (x.p \land x.f - 1) / fforder(x)$

ispowerful(x)

True (1) if x is a powerful integer, false (0) if not; an integer is powerful if and only if its valuation at all primes dividing x is greater than 1.

? ispowerful(50) %1 = 0 ? ispowerful(100) %2 = 1 ? ispowerful(5^3*(10^1000+1)^2) %3 = 1

isprime(x, flag)

True (1) if x is a prime number, false (0) otherwise. A prime number is a positive integer having exactly two distinct divisors among the natural numbers, namely 1 and itself.

This routine proves or disproves rigorously that a number is prime, which can be very slow when x is indeed a large prime integer. For instance a 1000 digits prime should require 15 to 30 minutes with default algorithms. Use ispseudoprime to quickly check for compositeness. Use primecert in order to obtain a primality proof instead of a yes/no answer; see also factor.

The function accepts vector/matrices arguments, and is then applied componentwise.

If flag = 0, use a combination of

- Baillie-Pomerance-Selfridge-Wagstaff compositeness test (see ispseudoprime),
- Selfridge "p-1" test if x-1 is smooth enough,
- Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) for general medium-sized x (less than 1500 bits),
- Atkin-Morain's Elliptic Curve Primality Prover (ECPP) for general large x.

If flag = 1, use Selfridge-Pocklington-Lehmer "p - 1" test; this requires partially factoring various auxilliary integers and is likely to be very slow.

If flag = 2, use APRCL only.

If flag = 3, use ECPP only.

isprimepower(x, n)

If $x = p^k$ is a prime power (p prime, k > 0), return k, else return 0. If a second argument n is given and x is indeed the k-th power of a prime p, sets n to p.

ispseudoprime(x, flag)

True (1) if x is a strong pseudo prime (see below), false (0) otherwise. If this function returns false, x is not prime; if, on the other hand it returns true, it is only highly likely that x is a prime number. Use *isprime* (which is of course much slower) to prove that x is indeed prime. The function accepts vector/matrices arguments, and is then applied componentwise.

If flag = 0, checks whether x has no small prime divisors (up to 101 included) and is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime. Such a pseudo prime passes a Rabin-Miller test for base 2, followed by a Lucas test for the sequence (P, 1), where $P \ge 3$ is the smallest odd integer such that $P^2 - 4$ is not a square mod x. (Technically, we are using an "almost extra strong Lucas test" that checks whether V_n is 2, without computing U_{n} .)

There are no known composite numbers passing the above test, although it is expected that infinitely many such numbers exist. In particular, all composites $\leq 2^{64}$ are correctly detected (checked using http://www.cecm. sfu.ca/Pseudoprimes/index-2-to-64.html).

If flag > 0, checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

ispseudoprimepower(x, n)

If $x = p^k$ is a pseudo-prime power (p pseudo-prime as per ispseudoprime, k > 0), return k, else return 0. If a second argument n is given and x is indeed the k-th power of a prime p, sets n to p.

More precisely, k is always the largest integer such that $x = n^k$ for some integer n and, when $n <= 2^{64}$ the function returns k > 0 if and only if n is indeed prime. When $n > 2^{64}$ is larger than the threshold, the function may return 1 even though n is composite: it only passed an ispseudoprime(n) test.

issquare(x, n)

True (1) if x is a square, false (0) if not. What "being a square" means depends on the type of x: all t_COMPLEX are squares, as well as all nonnegative t_REAL; for exact types such as t_INT, t_FRAC and t_INTMOD, squares are numbers of the form s^2 with s in \mathbb{Z} , \mathbb{Q} and $\mathbb{Z}/N\mathbb{Z}$ respectively.

```
? issquare(3) \\ as an integer
%1 = 0
? issquare(3.) \\ as a real number
%2 = 1
? issquare(Mod(7, 8)) \\ in Z/8Z
%3 = 0
? issquare( 5 + 0(13<sup>4</sup>) ) \\ in Q_13
%4 = 0
```

If n is given, a square root of x is put into n.

```
? issquare(4, &n)
%1 = 1
? n
%2 = 2
```

For polynomials, either we detect that the characteristic is 2 (and check directly odd and even-power monomials) or we assume that 2 is invertible and check whether squaring the truncated power series for the square root yields the original input.

For t_POLMOD x, we only support t_POLMOD s of t_INTMOD s encoding finite fields, assuming without checking that the intmod modulus p is prime and that the polmod modulus is irreducible modulo p.

```
? issquare(Mod(Mod(2,3), x^2+1), &n)
%1 = 1
? n
%2 = Mod(Mod(2, 3)*x, Mod(1, 3)*x^2 + Mod(1, 3))
```

issquarefree(x)

True (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial with coefficients in an integral domain.

```
? issquarefree(12)
%1 = 0
? issquarefree(6)
%2 = 1
? issquarefree(x^3+x^2)
%3 = 0
? issquarefree(Mod(1,4)*(x^2+x+1)) \\ Z/4Z is not a domain !
*** at top-level: issquarefree(Mod(1,4)*(x^2+x+1))
*** ^------
*** issquarefree: impossible inverse in Fp_inv: Mod(2, 4).
```

A polynomial is declared squarefree if gcd(x, x') is 1. In particular a nonzero polynomial with inexact coefficients is considered to be squarefree. Note that this may be inconsistent with factor, which first rounds the input to some exact approximation before factoring in the appropriate domain; this is correct when the input is not close to an inseparable polynomial (the resultant of x and x' is not close to 0).

An integer can be input in factored form as in arithmetic functions.

```
? issquarefree(factor(6))
%1 = 1
\\ count squarefree integers up to 10^8
? c = 0; for(d = 1, 10^8, if (issquarefree(d), c++)); c
time = 3min, 2,590 ms.
%2 = 60792694
? c = 0; forfactored(d = 1, 10^8, if (issquarefree(d), c++)); c
time = 45,348 ms. \\ faster !
%3 = 60792694
```

istotient(x, N)

True (1) if $x = \phi(n)$ for some integer n, false (0) if not.

? istotient(14)
%1 = 0
? istotient(100)
%2 = 0

If N is given, set N = n as well.

```
? istotient(4, &n)
%1 = 1
? n
%2 = 10
```

kill(sym)

Restores the symbol sym to its "undefined" status, and deletes any help messages attached to sym using addhelp. Variable names remain known to the interpreter and keep their former priority: you cannot make a variable "less important" by killing it!

? z = y = 1; y
%1 = 1
? kill(y)
? y \\ restored to ``undefined'' status
%2 = y
? variable()
%3 = [x, y, z] \\ but the variable name y is still known, with y > z !

For the same reason, killing a user function (which is an ordinary variable holding a t_CLOSURE) does not remove its name from the list of variable names.

If the symbol is attached to a variable — user functions being an important special case —, one may use the quote operator a = 'a to reset variables to their starting values. However, this will not delete a help message attached to a, and is also slightly slower than kill(a).

```
? x = 1; addhelp(x, "foo"); x
%1 = 1
? x = 'x; x \\ same as 'kill', except we don't delete help.
```
%2 = x ? ?x foo

On the other hand, kill is the only way to remove aliases and installed functions.

```
? alias(fun, sin);
? kill(fun);
? install(addii, GG);
? kill(addii);
```

kronecker(x, y)

Kronecker symbol (x||y), where x and y must be of type integer. By definition, this is the extension of Legendre symbol to $\mathbb{Z}x\mathbb{Z}$ by total multiplicativity in both arguments with the following special rules for y = 0, -1 or 2:

- (x||0) = 1 if ||x|| = 1 and 0 otherwise.
- $(x \parallel -1) = 1$ if $x \ge 0$ and -1 otherwise.
- (x||2) = 0 if x is even and 1 if $x = 1, -1 \mod 8$ and -1 if $x = 3, -3 \mod 8$.

lambertw(y, precision)

Lambert W function, solution of the implicit equation $xe^x = y$, for a positive real number y. This is the restriction to the positive reals of the complex principal branch W_0 , which is not implemented outside of \mathbb{R}^*_+ . Other branches W_k for k! = 0 are not implemented either.

laurentseries(f, serprec, M, precision)

Expand f as a Laurent series around x = 0 to order M. This function computes $f(x + O(x^n))$ until n is large enough: it must be possible to evaluate f on a power series with 0 constant term.

Note that individual Laurent coefficients of order $\leq M$ can be retrieved from s = laurentseries(f, M) via polcoef(s,i) for any $i \leq M$. The series s may occasionally be more precise that the required $O(x^{M+1})$.

With respect to successive calls to derivnum, laurentseries is both faster and more precise:

```
? exp(2) * cos(exp(2));
%6 = 3.3129294231043339804683687620360224365
? derivnum(x = 2, f(x))
%7 = 3.3129294231043339804683687620360224364 \\ 1 ulp off
? default(realprecision,115);
? for(i=1,10^4, laurentseries(x->f(x+2),1))
time = 279 ms.
? for(i=1,10^4, derivnum(x=2,f(x))) \\ ... and slower
time = 1,134 ms.
```

lcm(x, y)

Least common multiple of x and y, i.e. such that lcm(x, y) * gcd(x, y) = x * y, up to units. If y is omitted and x is a vector, returns the *lcm* of all components of x. For integer arguments, return the nonnegative lcm.

When x and y are both given and one of them is a vector/matrix type, the LCM is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to lcm(x, y[i]), resp. lcm(x, y[,i]). Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, lcm is not commutative.

Note that lcm(v) is quite different from

l = v[1]; for (i = 1, #v, l = lcm(l, v[i]))

Indeed, lcm(v) is a scalar, but 1 may not be (if one of the v[i] is a vector/matrix). The computation uses a divide-conquer tree and should be much more efficient, especially when using the GMP multiprecision kernel (and more subquadratic algorithms become available):

```
? v = vector(10^5, i, random);
? lcm(v);
time = 546 ms.
? l = v[1]; for (i = 1, #v, l = lcm(l, v[i]))
time = 4,561 ms.
```

length(x)

Length of x; #x is a shortcut for length(x). This is mostly useful for

- vectors: dimension (0 for empty vectors),
- lists: number of entries (0 for empty lists),
- maps: number of entries (0 for empty maps),
- matrices: number of columns,
- character strings: number of actual characters (without trailing 0, should you expect it from C char*).

```
? #"a string"
%1 = 8
? #[3,2,1]
%2 = 3
? #[]
%3 = 0
? #matrix(2,5)
%4 = 5
? L = List([1,2,3,4]); #L
```

%5 = 4 ? M = Map([a,b; c,d; e,f]); #M %6 = 3

The routine is in fact defined for arbitrary GP types, but is awkward and useless in other cases: it returns the number of non-code words in x, e.g. the effective length minus 2 for integers since the t_INT type has two code words.

lex(x, y)

Gives the result of a lexicographic comparison between x and y (as -1, 0 or 1). This is to be interpreted in quite a wide sense: it is admissible to compare objects of different types (scalars, vectors, matrices), provided the scalars can be compared, as well as vectors/matrices of different lengths; finally, when comparing two scalars, a complex number a + I * b is interpreted as a vector [a, b] and a real number a as [a, 0]. The comparison is recursive.

In case all components are equal up to the smallest length of the operands, the more complex is considered to be larger. More precisely, the longest is the largest; when lengths are equal, we have matrix > vector > scalar. For example:

```
? lex([1,3], [1,2,5])
%1 = 1
? lex([1,3], [1,3,-1])
%2 = -1
? lex([1], [[1]])
%3 = -1
? lex([1], [1]~)
%4 = 0
? lex(2 - I, 1)
%5 = 1
? lex(2 - I, 2)
%6 = 2
```

lfun(*L*, *s*, *D*, *precision*)

Compute the L-function value L(s), or if D is set, the derivative of order D at s. The parameter L is either an Lmath, an Ldata (created by lfuncreate, or an Linit (created by lfuninit), preferrably the latter if many values are to be computed.

The argument s is also allowed to be a power series; for instance, if $s = \alpha + x + O(x^n)$, the function returns the Taylor expansion of order n around α . The result is given with absolute error less than 2^{-B} , where B = realbit precision.

Caveat. The requested precision has a major impact on runtimes. It is advised to manipulate precision via realbitprecision as explained above instead of realprecision as the latter allows less granularity: realprecision increases by increments of 64 bits, i.e. 19 decimal digits at a time.

```
? lfun(x^2+1, 2) \\ Lmath: Dedekind zeta for Q(i) at 2
%1 = 1.5067030099229850308865650481820713960
? L = lfuncreate(ellinit("5077a1")); \\ Ldata: Hasse-Weil zeta function
? lfun(L, 1+x+0(x^4)) \\ zero of order 3 at the central point
%3 = 0.E-58 - 5.[...] E-40*x + 9.[...] E-40*x^2 + 1.7318[...]*x^3 + 0(x^4)
\\ Linit: zeta(1/2+it), |t| < 100, and derivative
? L = lfuninit(1, [100], 1);
? T = lfunzeros(L, [1,25]);
```

```
%5 = [14.134725[...], 21.022039[...]]
? z = 1/2 + I*T[1];
? abs( lfun(L, z) )
%7 = 8.7066865533412207420780392991125136196 E-39
? abs( lfun(L, z, 1) )
%8 = 0.79316043335650611601389756527435211412 \\ simple zero
```

lfunabelianrelinit(*bnfL*, *bnfK*, *polrel*, *sdom*, *der*, *precision*)

Returns the Linit structure attached to the Dedekind zeta function of the number field L (see lfuninit), given a subfield K such that L/K is abelian. Here polrel defines L over K, as usual with the priority of the variable of bnfK lower than that of polrel. sdom and der are as in lfuninit.

```
? D = -47; K = bnfinit(y^2-D);
? rel = quadhilbert(D); T = rnfequation(K.pol, rel); \\ degree 10
? L = lfunabelianrelinit(T,K,rel, [2,0,0]); \\ at 2
time = 84 ms.
? lfun(L, 2)
%4 = 1.0154213394402443929880666894468182650
? lfun(T, 2) \\ using parisize > 300MB
time = 652 ms.
%5 = 1.0154213394402443929880666894468182656
```

As the example shows, using the (abelian) relative structure is more efficient than a direct computation. The difference becomes drastic as the absolute degree increases while the subfield degree remains constant.

lfunan(*L*, *n*, *precision*)

Compute the first *n* terms of the Dirichlet series attached to the *L*-function given by L (Lmath, Ldata or Linit).

```
? lfunan(1, 10) \\ Riemann zeta
%1 = [1, 1, 1, 1, 1, 1, 1, 1, 1]
? lfunan(5, 10) \\ Dirichlet L-function for kronecker(5,.)
%2 = [1, -1, -1, 1, 0, 1, -1, -1, 1, 0]
```

lfunartin(*nf*, *gal*, *rho*, *n*, *precision*)

Returns the Ldata structure attached to the Artin L-function provided by the representation ρ of the Galois group of the extension K/\mathbb{Q} , defined over the cyclotomic field $\mathbb{Q}(\zeta_n)$, where *nf* is the nfinit structure attached to *K*, *gal* is the galoisinit structure attached to K/\mathbb{Q} , and *rho* is given either

- by the values of its character on the conjugacy classes (see galoisconjclasses and galoischartable)
- or by the matrices that are the images of the generators : emphasis: `gal.gen`.

Cyclotomic numbers in rho are represented by polynomials, whose variable is understood as the complex number $\exp(2i\pi/n)$.

In the following example we build the Artin L-functions attached to the two irreducible degree 2 representations of the dihedral group D_{10} defined over $\mathbb{Q}(\zeta_5)$, for the extension H/\mathbb{Q} where H is the Hilbert class field of $\mathbb{Q}(\sqrt{-47})$. We show numerically some identities involving Dedekind ζ functions and Hecke L series.

```
? P = quadhilbert(-47)
%1 = x^5 + 2*x^4 + 2*x^3 + x^2 - 1
? N = nfinit(nfsplitting(P));
? G = galoisinit(N); \\ D_10
? [T,n] = galoischartable(G);
```

```
? T \\ columns give the irreducible characters
%5 =
[1 1 2 2]
[1 - 1 \ 0 \ 0]
[1 \ 1 \ -y^3 \ -y^2 \ -1 \ y^3 \ +y^2]
\begin{bmatrix} 1 & 1 & y^3 + y^2 & -y^3 & -y^2 & -1 \end{bmatrix}
? n
\%6 = 5
? L2 = lfunartin(N,G, T[,2], n);
? L3 = lfunartin(N,G, T[,3], n);
? L4 = lfunartin(N,G, T[,4], n);
? s = 1 + x + 0(x^4);
? lfun(-47,s) - lfun(L2,s)
%11 ~ 0
? lfun(1,s)*lfun(-47,s)*lfun(L3,s)^2*lfun(L4,s)^2 - lfun(N,s)
%12 ~ 0
? lfun(1,s)*lfun(L3,s)*lfun(L4,s) - lfun(P,s)
%13 ~ 0
? bnr = bnrinit(bnfinit(x^{2}+47),1,1);
? bnr.cvc
%15 = [5] \setminus Z/5Z: 4 nontrivial ray class characters
? lfun([bnr,[1]], s) - lfun(L3, s)
%16 ~ 0
? lfun([bnr,[2]], s) - lfun(L4, s)
%17 ~ 0
? lfun([bnr,[3]], s) - lfun(L3, s)
%18 ~ 0
? lfun([bnr,[4]], s) - lfun(L4, s)
%19 ~ 0
```

The first identify identifies the nontrivial abelian character with (-47, .); the second is the factorization of the regular representation of D_{10} ; the third is the factorization of the natural representation of $D_{10} \subset S_5$; and the final four are the expressions of the degree 2 representations as induced from degree 1 representations.

lfuncheckfeq(*L*, *t*, *precision*)

Given the data attached to an *L*-function (Lmath, Ldata or Linit), check whether the functional equation is satisfied. This is most useful for an Ldata constructed "by hand", via lfuncreate, to detect mistakes.

If the function has poles, the polar part must be specified. The routine returns a bit accuracy b such that $||w - w|| < 2^b$, where w is the root number contained in data, and

$$w = \theta(1/t)t^{-k}/\overline{\theta}(t)$$

is a computed value derived from the assumed functional equation. If the parameter t is omitted, we try random samples on the real line in the segment [1, 1.25]. Of course, a large negative value of the order of realbitprecision is expected but if $\overline{\theta}$ is very small all over the sampled segment, you should first increase realbitprecision by $-\log_2 \|\overline{\theta}(t)\|$ (which is positive if θ is small) to get a meaningful result.

If t is given, it should be close to the unit disc for efficiency and such that $\overline{\theta}(t)! = 0$. We then check the functional equation at that t. Again, if $\overline{\theta}(t)$ is very small, you should first increase realbitprecision to get a useful result.

```
? \pb 128 \\ 128 bits of accuracy
? default(realbitprecision)
%1 = 128
? L = lfuncreate(1); \\ Riemann zeta
? lfuncheckfeq(L)
%3 = -124
```

i.e. the given data is consistent to within 4 bits for the particular check consisting of estimating the root number from all other given quantities. Checking away from the unit disc will either fail with a precision error, or give disappointing results (if $\theta(1/t)$ is large it will be computed with a large absolute error)

```
? lfuncheckfeq(L, 2+I)
%4 = -115
? lfuncheckfeq(L,10)
*** at top-level: lfuncheckfeq(L,10)
*** ^------
*** lfuncheckfeq: precision too low in lfuncheckfeq.
```

lfunconductor(*L*, *setN*, *flag*, *precision*)

Compute the conductor of the given *L*-function (if the structure contains a conductor, it is ignored). Two methods are available, depending on what we know about the conductor, encoded in the setN parameter:

• setN is a scalar: we know nothing but expect that the conductor lies in the interval [1, setN].

If flag is 0 (default), give either the conductor found as an integer, or a vector (possibly empty) of conductors found. If flag is 1, same but give the computed floating point approximations to the conductors found, without rounding to integers. It flag is 2, give all the conductors found, even those far from integers.

Caveat. This is a heuristic program and the result is not proven in any way:

```
? L = lfuncreate(857); \\ Dirichlet L function for kronecker(857,.)
? \p19
realprecision = 19 significant digits
? lfunconductor(L)
%2 = [17, 857]
? lfunconductor(L,,1) \\ don't round
%3 = [16.99999999999999999, 857.000000000000000]
? \p38
realprecision = 38 significant digits
? lfunconductor(L)
%4 = 857
```

Increasing setN or increasing realbitprecision slows down the program but gives better accuracy for the result. This algorithm should only be used if the primes dividing the conductor are unknown, which is uncommon.

• setN is a vector of possible conductors; for instance of the form D1 * divisors(D2), where D_1 is the known part of the conductor and D_2 is a multiple of the contribution of the bad primes.

In that case, flag is ignored and the routine uses lfuncheckfeq. It returns [N, e] where N is the best conductor in the list and e is the value of lfuncheckfeq for that N. When no suitable conductor exist or there is a tie among best potential conductors, return the empty vector [].

```
? E = ellinit([0,0,0,4,0]); /* Elliptic curve y^2 = x^3+4x */
? E.disc \\ |disc E| = 2^12
```

```
\%2 = -4096
\backslash create Ldata by hand. Guess that root number is 1 and conductor N
? L(N) = lfuncreate([n->ellan(E,n), 0, [0,1], 2, N, 1]);
\\ lfunconductor ignores conductor = 1 in Ldata !
? lfunconductor(L(1), divisors(E.disc))
\%5 = [32, -127]
? fordiv(E.disc, d, print(d,": ",lfuncheckfeq(L(d)))) \\ direct check
1: 0
2: 0
4: -1
8: -2
16: -3
32: -127
64: -3
128: -2
256: -2
512: -1
1024: -1
2048: 0
4096: 0
```

The above code assumed that root number was 1; had we set it to -1, none of the lfuncheckfeq values would have been acceptable:

```
? L2 = lfuncreate([n->ellan(E,n), 0, [0,1], 2, 0, -1]);
? lfunconductor(L2, divisors(E.disc))
%7 = []
```

lfuncost(*L*, *sdom*, *der*, *precision*)

Estimate the cost of running lfuninit(L, sdom, der) at current bit precision. Returns [t, b], to indicate that t coefficients a_n will be computed, as well as t values of gammamellininv, all at bit accuracy b. A subsequent call to lfun at s evaluates a polynomial of degree t at exp(hs) for some real parameter h, at the same bit accuracy b. If L is already an Linit, then *sdom* and *der* are ignored and are best left omitted; the bit accuracy is also inferred from L: in short we get an estimate of the cost of using that particular Linit.

? \pb 128

```
? lfuncost(1, [100]) \\ for zeta(1/2+I*t), |t| < 100
%1 = [7, 242] \\ 7 coefficients, 242 bits
? lfuncost(1, [1/2, 100]) \\ for zeta(s) in the critical strip, |Im s| < 100
%2 = [7, 246] \\ now 246 bits
? lfuncost(1, [100], 10) \\ for zeta(1/2+I*t), |t| < 100
%3 = [8, 263] \\ 10th derivative increases the cost by a small amount
? lfuncost(1, [10^5])
%3 = [158, 113438] \\ larger imaginary part: huge accuracy increase
? L = lfuncreate(polcyclo(5)); \\ Dedekind zeta for Q(zeta_5)
? lfuncost(L, [100]) \\ at s = 1/2+I*t), |t| < 100
%5 = [11457, 582]
? lfuncost(L, [200]) \\ twice higher
%6 = [36294, 1035]
? lfuncost(L, [10^4]) \\ much higher: very costly !
%7 = [70256473, 45452]
```

```
? \pb 256
? lfuncost(L, [100]); \\ doubling bit accuracy
%8 = [17080, 710]
```

In fact, some L functions can be factorized algebraically by the lfuninit call, e.g. the Dedekind zeta function of abelian fields, leading to much faster evaluations than the above upper bounds. In that case, the function returns a vector of costs as above for each individual function in the product actually evaluated:

```
? L = lfuncreate(polcyclo(5)); \\ Dedekind zeta for Q(zeta_5)
? lfuncost(L, [100]) \\ a priori cost
%2 = [11457, 582]
? L = lfuninit(L, [100]); \\ actually perform all initializations
? lfuncost(L)
%4 = [[16, 242], [16, 242], [7, 242]]
```

The Dedekind function of this abelian quartic field is the product of four Dirichlet L-functions attached to the trivial character, a nontrivial real character and two complex conjugate characters. The nontrivial characters happen to have the same conductor (hence same evaluation costs), and correspond to two evaluations only since the two conjugate characters are evaluated simultaneously. For a total of three L-functions evaluations, which explains the three components above. Note that the actual cost is much lower than the a priori cost in this case.

lfuncreate(obj)

This low-level routine creates Ldata structures, needed by *lfun* functions, describing an *L*-function and its functional equation. We advise using a high-level constructor when one is available, see ??lfun, and this function accepts them:

? L = lfuncreate(1); \\ Riemann zeta ? L = lfuncreate(5); \\ Dirichlet L-function for quadratic character (5/.) ? L = lfuncreate(x^2+1); \\ Dedekind zeta for Q(i) ? L = lfuncreate(ellinit([0,1])); \\ L-function of E/Q: y^2=x^3+1

One can then use, e.g., lfun(L, s) to directly evaluate the respective L-functions at s, or lfuninit(L, [c,w,h]) to initialize computations in the rectangular box $\Re(s-c) \le w$, $\Im(s) \le h$.

We now describe the low-level interface, used to input nonbuiltin L-functions. The input is now a 6 or 7 component vector V = [a, astar, Vga, k, N, eps, poles], whose components are as follows:

V[1] = a encodes the Dirichlet series coefficients (a_n). The preferred format is a closure of arity 1: n- > vector(n,i,a(i)) giving the vector of the first n coefficients. The closure is allowed to return a vector of more than n coefficients (only the first n will be considered) or even less than n, in which case loss of accuracy will occur and a warning that #an is less than expected is issued. This allows to precompute and store a fixed large number of Dirichlet coefficients in a vector v and use the closure n- > v, which does not depend on n. As a shorthand for this latter case, you can input the vector v itself instead of the closure.

```
? z = lfuncreate([n->vector(n,i,1), 1, [0], 1, 1, 1, 1]); \\ Riemann zeta
? lfun(z,2) - Pi^2/6
%2 = -5.877471754111437540 E-39
```

A second format is limited to L-functions affording an Euler product. It is a closure of arity 2 (p,d)- > F(p) giving the local factor $L_p(X)$ at p as a rational function, to be evaluated at p^{-s} as in direuler; d is set to logint(n, p) + 1, where n is the total number of Dirichlet coefficients $(a_1, ..., a_n)$ that will be computed. In other words, the smallest integer d such that $p^d > n$. This parameter d allows to compute only part of L_p when p is large and L_p expensive to compute: any polynomial (or t_SER) congruent to L_p modulo X^d is acceptable since only the coefficients of $X^0, ..., X^{d-1}$ are needed to expand the Dirichlet series. The closure can of course ignore this parameter:

? z = lfuncreate([(p,d)->1/(1-x), 1, [0], 1, 1, 1, 1]); \\ Riemann zeta ? lfun(z,2) - Pi^2/6 %4 = -5.877471754111437540 E-39

One can describe separately the generic local factors coefficients and the bad local factors by setting $dir = [F, L_{bad}]$, were $L_{bad} = [[p_1, L_{p_1}], ..., [p_k, L_{p_k}]]$, where F describes the generic local factors as above, except that when $p = p_i$ for some i <= k, the coefficient a_p is directly set to L_{p_i} instead of calling F.

N = 15; E = ellinit([1, 1, 1, -10, -10]); \\ = "15a1" F(p,d) = 1 / (1 - ellap(E,p)*'x + p*'x^2); Lbad = [[3, 1/(1+'x)], [5, 1/(1-'x)]]; L = lfuncreate([[F,Lbad], 0, [0,1], 2, N, ellrootno(E)]);

Of course, in this case, lfuncreate(E) is preferable!

- V[2] = astar is the Dirichlet series coefficients of the dual function, encoded as a above. The sentinel values 0 and 1 may be used for the special cases where $a = a^*$ and $a = \overline{a^*}$, respectively.
- V[3] = Vga is the vector of α_j such that the gamma factor of the L-function is equal to

where : math : $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2} \Gamma(s/2)$. This same syntaxis used in the : literal : 'gamma mellinin' functions. In particular the set of the set o

• V[4] = k is a positive integer k. The functional equation relates values at s and k - s. For instance, for an Artin L-series such as a Dedekind zeta function we have k = 1, for an elliptic curve k = 2, and for a modular form, k is its weight. For motivic L-functions, the *motivic* weight w is w = k - 1.

By default we assume that $a_n = O_{\epsilon}(n^{k_1+\epsilon})$, where $k_1 = w$ and even $k_1 = w/2$ when the L function has no pole (Ramanujan-Petersson). If this is not the case, you can replace the k argument by a vector $[k, k_1]$, where k_1 is the upper bound you can assume.

- V[5] = N is the conductor, an integer $N \ge 1$, such that $\Lambda(s) = N^{s/2} \gamma_A(s) L(s)$ with $\gamma_A(s)$ as above.
- V[6] = eps is the root number ε , i.e., the complex number (usually of modulus 1) such that $\Lambda(a, k s) = \varepsilon \Lambda(a^*, s)$.
- The last optional component V[7] = poles encodes the poles of the L or Λ-functions, and is omitted if they have no poles. A polar part is given by a list of 2-component vectors [β, P_β(x)], where β is a pole and the power series P_β(x) describes the attached polar part, such that L(s) P_β(s β) is holomorphic in a neighbourhood of β. For instance P_β = r/x + O(1) for a simple pole at β or r₁/x² + r₂/x + O(1) for a double pole. The type of the list describing the polar part allows to distinguish between L and A: a t_VEC is attached to L, and a t_COL is attached to Λ. Unless a = a^{*} (coded by astar equal to 0 or 1), it is mandatory to specify the polar part of Λ rather than those of L since the poles of L* cannot be infered from the latter ! Whereas the functional equation allows to deduce the polar part of Λ* from the polar part of Λ.

Finally, if $a = \overline{a^*}$, we allow a shortcut to describe the frequent situation where L has at most simple pole, at s = k, with residue r a complex scalar: you may then input poles = r. This value r can be set to 0 if unknown and it will be computed.

When one component is not exact. Alternatively, obj can be a closure of arity 0 returning the above vector to the current real precision. This is needed if some components are not available exactly but only through floating point approximations. The closure allows algorithms to recompute them to higher accuracy when needed. Compare

? Ld1() = [n->lfunan(Mod(2,7),n),1,[0],1,7,((-13-3*sqrt(-3))/14)^(1/6)]; ? Ld2 = [n->lfunan(Mod(2,7),n),1,[0],1,7,((-13-3*sqrt(-3))/14)^(1/6)]; ? L1 = lfuncreate(Ld1); ? L2 = lfuncreate(Ld2); ? lfun(L1,1/2+I*200) \\ OK %5 = 0.55943925130316677665287870224047183265 -0.42492662223174071305478563967365980756*I ? lfun(L2,1/2+I*200) \\ all accuracy lost %6 = 0.E-38 + 0.E-38*I

The accuracy lost in Ld2 is due to the root number being given to an insufficient precision. To see what happens try

```
? Ld3() = printf("prec needed: %ld bits",getlocalbitprec());Ld1()
? L3 = lfuncreate(Ld3);
prec needed: 64 bits
? z3 = lfun(L3,1/2+I*200)
prec needed: 384 bits
%16 = 0.55943925130316677665287870224047183265 -
0.42492662223174071305478563967365980756*I
```

lfundiv(*L1*, *L2*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the quotient of the Dirichlet series L_1 and L_2 given by L1 and L2. Assume that $v_z(L_1) \ge v_z(L_2)$ at all complex numbers z: the construction may not create new poles, nor increase the order of existing ones.

lfundual(*L*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the dual L-function ^L of L. If k and ε are respectively the weight and root number of L, then the following formula holds outside poles, up to numerical errors:

$$\Lambda(L,s) = \varepsilon \Lambda(^L, k-s).$$

? L = lfunqf(matdiagonal([1,2,3,4])); ? eps = lfunrootres(L)[3]; k = L[4]; ? M = lfundual(L); lfuncheckfeq(M) %3 = -127 ? s= 1+Pi*I; ? a = lfunlambda(L,s); ? b = eps * lfunlambda(M,k-s); ? exponent(a - b) %7 = -130

lfunetaquo(M)

Returns the Ldata structure attached to the L function attached to the modular form $z : --- > \prod_{i=1}^{n} \eta(M_{i,1}z)^{M_{i,2}}$ It is currently assumed that f is a self-dual cuspidal form on $\Gamma_0(N)$ for some N. For instance, the L-function $\sum \tau(n)n^{-s}$ attached to Ramanujan's Δ function is encoded as follows

```
? L = lfunetaquo(Mat([1,24]));
? lfunan(L, 100) \\ first 100 values of tau(n)
```

For convenience, a t_VEC is also accepted instead of a factorization matrix with a single row:

? L = lfunetaquo([1,24]); \\ same as above

lfungenus2(F)

Returns the Ldata structure attached to the L function attached to the genus-2 curve defined by $y^2 = F(x)$ or $y^2 + Q(x)y = P(x)$ if F = [P,Q]. Currently, the model needs to be minimal at 2, and if the conductor is even, its valuation at 2 might be incorrect (a warning is issued).

lfunhardy(*L*, *t*, *precision*)

Variant of the Hardy Z-function given by L, used for plotting or locating zeros of L(k/2+it) on the critical line. The precise definition is as follows: let k/2 be the center of the critical strip, d be the degree, $Vga = (\alpha_j)_{j < =d}$ given the gamma factors, and ε be the root number; we set $s = k/2 + it = \rho e^{i\theta}$ and $2E = d(k/2-1) + \Re(\sum_{1 \le i \le d} \alpha_j)$. Assume first that Λ is self-dual, then the computed function at t is equal to

$$Z(t) = \varepsilon^{-1/2} \Lambda(s) \cdot \rho^{-E} e^{dt\theta/2},$$

which is a real function of t vanishing exactly when L(k/2+it) does on the critical line. The normalizing factor $||s||^{-E}e^{dt\theta/2}$ compensates the exponential decrease of $\gamma_A(s)$ as $t \to oo$ so that Z(t) 1. For non-self-dual Λ , the definition is the same except we drop the $\varepsilon^{-1/2}$ term (which is not well defined since it depends on the chosen dual sequence $a^*(n)$): Z(t) is still of the order of 1 and still vanishes where L(k/2+it) does, but it needs no longer be real-valued.

? T = 100; \\ maximal height
? L = lfuninit(1, [T]); \\ initialize for zeta(1/2+it), |t|<T
? \p19 \\ no need for large accuracy
? ploth(t = 0, T, lfunhardy(L,t))</pre>

Using lfuninit is critical for this particular applications since thousands of values are computed. Make sure to initialize up to the maximal t needed: otherwise expect to see many warnings for unsufficient initialization and suffer major slowdowns.

lfuninit(*L*, *sdom*, *der*, *precision*)

Initialization function for all functions linked to the computation of the *L*-function L(s) encoded by L, where *s* belongs to the rectangular domain sdom = [center, w, h] centered on the real axis, $||\Re(s) - center|| \le w$, $||\Im(s)|| \le h$, where all three components of sdom are real and *w*, *h* are nonnegative. der is the maximum order of derivation that will be used. The subdomain [k/2, 0, h] on the critical line (up to height *h*) can be encoded as [h] for brevity. The subdomain [k/2, w, h] centered on the critical line can be encoded as [w, h] for brevity.

The argument L is an Lmath, an Ldata or an Linit. See ??Ldata and ??lfuncreate for how to create it.

The height h of the domain is a *crucial* parameter: if you only need L(s) for real s, set h to 0. The running time is roughly proportional to

$$(B/d + \pi h/4)^{d/2+3}N^{1/2}$$

where B is the default bit accuracy, d is the degree of the L-function, and N is the conductor (the exponent d/2+3 is reduced to d/2 + 2 when d = 1 and d = 2). There is also a dependency on w, which is less crucial, but make sure to use the smallest rectangular domain that you need.

```
? L0 = lfuncreate(1); \\ Riemann zeta
? L = lfuninit(L0, [1/2, 0, 100]); \\ for zeta(1/2+it), |t| < 100
? lfun(L, 1/2 + I)
? L = lfuninit(L0, [100]); \\ same as above !
```

lfunlambda(L, s, D, precision)

Compute the completed L-function $\Lambda(s) = N^{s/2}\gamma(s)L(s)$, or if D is set, the derivative of order D at s. The parameter L is either an Lmath, an Ldata (created by lfuncreate, or an Linit (created by lfuninit), preferrably the latter if many values are to be computed.

The result is given with absolute error less than $2^{-B} \|\gamma(s) N^{s/2}\|$, where B = real bit precision.

lfunmf(*mf*, *F*, *precision*)

If F is a modular form in mf, output the L-functions corresponding to its $[\mathbb{Q}(F) : \mathbb{Q}(\chi)]$ complex embeddings, ready for use with the lfun package. If F is omitted, output the L-functions attached to all eigenforms in the new space; the result is a vector whose length is the number of Galois orbits of newforms. Each entry contains the vector of L-functions corresponding to the d complex embeddings of an orbit of dimension d over $\mathbb{Q}(\chi)$.

```
? mf = mfinit([35,2],0);mffields(mf)
%1 = [y, y^2 - y - 4]
? f = mfeigenbasis(mf)[2]; mfparams(f) \\ orbit of dimension two
%2 = [35, 2, 1, y^2 - y - 4, t - 1]
? [L1,L2] = lfunmf(mf, f); \\ Two L-functions
? lfun(L1,1)
%4 = 0.81018461849460161754947375433874745585
? lfun(L2,1)
%5 = 0.46007635204895314548435893464149369804
? [ lfun(L,1) | L <- concat(lfunmf(mf)) ]
%6 = [0.70291..., 0.81018..., 0.46007...]
```

The concat instruction concatenates the vectors corresponding to the various (here two) orbits, so that we obtain the vector of all the *L*-functions attached to eigenforms.

lfunmfspec(L, precision)

Let L be the L-function attached to a modular eigenform f of weight k, as given by lfunmf. In even weight, returns [ve,vo,om,op], where ve (resp., vo) is the vector of even (resp., odd) periods of f and om and op the corresponding real numbers ω^- and ω^+ normalized in a noncanonical way. In odd weight ominus is the same as op and we return [v,op] where v is the vector of all periods.

```
? D = mfDelta(); mf = mfinit(D); L = lfunmf(mf, D);
? [ve, vo, om, op] = lfunmfspec(L)
%2 = [[1, 25/48, 5/12, 25/48, 1], [1620/691, 1, 9/14, 9/14, 1, 1620/691],\
0.0074154209298961305890064277459002287248,\
0.0050835121083932868604942901374387473226]
? DS = mfsymbol(mf, D); bestappr(om*op / mfpetersson(DS), 10^8)
%3 = 8192/225
? mf = mfinit([4, 9, -4], 0);
? F = mfeigenbasis(mf)[1]; L = lfunmf(mf, F);
? [v, om] = lfunmfspec(L)
%6 = [[1, 10/21, 5/18, 5/24, 5/24, 5/18, 10/21, 1],\
1.1302643192034974852387822584241400608]
? FS = mfsymbol(mf, F); bestappr(om^2 / mfpetersson(FS), 10^8)
%7 = 113246208/325
```

lfunmul(*L1*, *L2*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the product of the Dirichlet series given by L1 and L2.

lfunorderzero(*L*, *m*, *precision*)

Computes the order of the possible zero of the L-function at the center k/2 of the critical strip; return 0 if L(k/2)

does not vanish.

If m is given and has a nonnegative value, assumes the order is at most m. Otherwise, the algorithm chooses a sensible default:

- if the L argument is an Linit, assume that a multiple zero at s = k/2 has order less than or equal to the maximal allowed derivation order.
- else assume the order is less than 4.

You may explicitly increase this value using optional argument *m*; this overrides the default value above. (Possibly forcing a recomputation of the Linit.)

lfunqf(*Q*, *precision*)

Returns the Ldata structure attached to the Θ function of the lattice attached to the primitive form proportional to the definite positive quadratic form Q.

```
? L = lfunqf(matid(2));
? lfunqf(L,2)
%2 = 6.0268120396919401235462601927282855839
? lfun(x^2+1,2)*4
%3 = 6.0268120396919401235462601927282855839
```

The following computes the Madelung constant:

```
? L1=lfunqf(matdiagonal([1,1,1]));
? L2=lfunqf(matdiagonal([4,1,1]));
? L3=lfunqf(matdiagonal([4,4,1]));
? F(s)=6*lfun(L2,s)-12*lfun(L3,s)-lfun(L1,s)*(1-8/4^s);
? F(1/2)
%5 = -1.7475645946331821906362120355443974035
```

lfunrootres(*data*, *precision*)

Given the Ldata attached to an *L*-function (or the output of lfunthetainit), compute the root number and the residues.

The output is a 3-component vector $[[[a_1, r_1], ..., [a_n, r_n], [[b_1, R_1], ..., [b_m, R_m]], w]$, where r_i is the polar part of L(s) at a_i , R_i is the polar part of $\Lambda(s)$ at b_i or [0, 0, r] if there is no pole, and w is the root number. In the present implementation,

• either the polar part must be completely known (and is then arbitrary): the function determines the root number,

```
? L = lfunmul(1,1); \\ zeta^2
? [r,R,w] = lfunrootres(L);
? r \\ single pole at 1, double
%3 = [[1, 1.[...]*x^-2 + 1.1544[...]*x^-1 + O(x^0)]]
? w
%4 = 1
? R \\ double pole at 0 and 1
%5 = [[1,[...]], [0,[...]]]~
```

• or at most a single pole is allowed: the function computes both the root number and the residue (0 if no pole).

lfunshift(*L*, *d*, *flag*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the shift of L by d, that is to the function L_d such that $L_d(s) = L(s - d)$. If flag = 1, return the product LxL_d instead.

? Z = lfuncreate(1); \\ zeta(s) ? L = lfunshift(Z,1); \\ zeta(s-1) ? normlp(Vec(lfunlambda(L,s)-lfunlambda(L,3-s))) %3 = 0.E-38 \\ the expansions coincide to 'seriesprecision' ? lfun(L,1) %4 = -0.500000000000000000000000000000000 \\ = zeta(0) ? M = lfunshift(Z,1,1); \\ zeta(s)*zeta(s-1) ? normlp(Vec(lfunlambda(M,s)-lfunlambda(M,2-s))) %6 = 2.350988701644575016 E-38 ? lfun(M,2) \\ simple pole at 2, residue zeta(2) %7 = 1.6449340668482264364724151666460251892*x^-1+0(x^0)

lfunsympow(E, m)

Returns the Ldata structure attached to the L function attached to the m-th symmetric power of the elliptic curve E defined over the rationals.

lfuntheta(data, t, m, precision)

Compute the value of the *m*-th derivative at t of the theta function attached to the L-function given by data. data can be either the standard L-function data, or the output of lfunthetainit. The result is given with absolute error less than 2^{-B} , where B = real bit precision.

The theta function is defined by the formula $\Theta(t) = \sum_{n>=1} a(n)K(nt/\sqrt{(N)})$, where a(n) are the coefficients of the Dirichlet series, N is the conductor, and K is the inverse Mellin transform of the gamma product defined by the Vga component. Its Mellin transform is equal to $\Lambda(s) - P(s)$, where $\Lambda(s)$ is the completed L-function and the rational function P(s) its polar part. In particular, if the L-function is the L-function of a modular form $f(\tau) = \sum_{n>=0} a(n)q^n$ with $q = \exp(2\pi i\tau)$, we have $\Theta(t) = 2(f(it/\sqrt{N}) - a(0))$. Note that a(0) = -L(f, 0) in this case.

lfunthetacost(*L*, *tdom*, *m*, *precision*)

This function estimates the cost of running lfunthetainit(L,tdom,m) at current bit precision. Returns the number of coefficients a_n that would be computed. This also estimates the cost of a subsequent evaluation lfuntheta, which must compute that many values of gammamellininv at the current bit precision. If L is already an Linit, then *tdom* and m are ignored and are best left omitted: we get an estimate of the cost of using that particular Linit.

```
? \pb 1000
? L = lfuncreate(1); \\ Riemann zeta
? lfunthetacost(L); \setminus cost for theta(t), t real >= 1
\%1 = 15
? lfunthetacost(L, 1 + I); \\ cost for theta(1+I). Domain error !
 *** at top-level: lfunthetacost(1,1+I)
 *** lfunthetacost: domain error in lfunthetaneed: arg t > 0.785
? lfunthetacost(L, 1 + I/2) \\ for theta(1+I/2).
%2 = 23
? lfunthetacost(L, 1 + I/2, 10) \\ for theta^((10))(1+I/2).
\%3 = 24
? lfunthetacost(L, [2, 1/10]) \\ cost for theta(t), |t| \ge 2, |arg(t)| < 1/10
%4 = 8
? L = lfuncreate( ellinit([1,1]) );
? lfunthetacost(L) \setminus for t >= 1
\%6 = 2471
```

lfunthetainit(*L*, *tdom*, *m*, *precision*)

Initialization function for evaluating the *m*-th derivative of theta functions with argument *t* in domain *tdom*. By default (*tdom* omitted), *t* is real, $t \ge 1$. Otherwise, *tdom* may be

- a positive real scalar ρ : t is real, $t \ge \rho$.
- a nonreal complex number: compute at this particular t; this allows to compute θ(z) for any complex z satisfying ||z|| >= ||t|| and || arg z|| <= || arg t||; we must have ||2 arg z/d|| < π/2, where d is the degree of the Γ factor.
- a pair [ρ, α]: assume that ||t|| >= ρ and || arg t|| ≤ α; we must have ||2α/d|| < π/2, where d is the degree of the Γ factor.

```
? \p500
? L = lfuncreate(1); \\ Riemann zeta
? t = 1+I/2;
? lfuntheta(L, t); \\ direct computation
time = 30 ms.
? T = lfunthetainit(L, 1+I/2);
time = 30 ms.
? lfuntheta(T, t); \\ instantaneous
```

The T structure would allow to quickly compute $\theta(z)$ for any z in the cone delimited by t as explained above. On the other hand

```
? lfuntheta(T,I)
 *** at top-level: lfuntheta(T,I)
 *** ^-----
 *** lfuntheta: domain error in lfunthetaneed: arg t > 0.785398163397448
```

The initialization is equivalent to

```
? lfunthetainit(L, [abs(t), arg(t)])
```

lfuntwist(*L*, *chi*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the twist of L by the primitive character attached to the Dirichlet character chi. The conductor of the character must be coprime to the conductor of the L-function L.

lfunzeros(*L*, *lim*, *divz*, *precision*)

lim being either a positive upper limit or a nonempty real interval, computes an ordered list of zeros of L(s) on the critical line up to the given upper limit or in the given interval. Use a naive algorithm which may miss some zeros: it assumes that two consecutive zeros at height $T \ge 1$ differ at least by $2\pi/\omega$, where

 $\omega := divz.(d\log(T/2\pi) + d + 2\log(N/(\pi/2)^d)).$

To use a finer search mesh, set divz to some integral value larger than the default (= 8).

```
? lfunzeros(1, 30) \\ zeros of Rieman zeta up to height 30
%1 = [14.134[...], 21.022[...], 25.010[...]]
? #lfunzeros(1, [100,110]) \\ count zeros with 100 <= Im(s) <= 110
%2 = 4</pre>
```

The algorithm also assumes that all zeros are simple except possibly on the real axis at s = k/2 and that there are no poles in the search interval. (The possible zero at s = k/2 is repeated according to its multiplicity.)

If you pass an Linit to the function, the algorithm assumes that a multiple zero at s = k/2 has order less than or equal to the maximal derivation order allowed by the Linit. You may increase that value in the Linit but this is costly: only do it for zeros of low height or in lfunorderzero instead.

lift(x, v)

If v is omitted, lifts intmods from $\mathbb{Z}/n\mathbb{Z}$ in \mathbb{Z} , p-adics from \mathbb{Q}_p to \mathbb{Q} (as truncate), and polmods to polynomials. Otherwise, lifts only polmods whose modulus has main variable v. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(lift,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

? lift(Mod(5,3))
%1 = 2
? lift(3 + 0(3^9))
%2 = 3
? lift(Mod(x,x^2+1))
%3 = x
? lift(Mod(x,x^2+1))
%4 = x

Lifts are performed recursively on an object components, but only by *one level*: once a t_POLMOD is lifted, the components of the result are *not* lifted further.

? lift(x * Mod(1,3) + Mod(2,3)) %4 = x + 2 ? lift(x * Mod(y,y^2+1) + Mod(2,3)) %5 = y*x + Mod(2, 3) \\ do you understand this one? ? lift(x * Mod(y,y^2+1) + Mod(2,3), 'x) %6 = Mod(y, y^2 + 1)*x + Mod(Mod(2, 3), y^2 + 1) ? lift(%, y) %7 = y*x + Mod(2, 3)

To recursively lift all components not only by one level, but as long as possible, use liftall. To lift only t_INTMOD s and t_PADIC s components, use liftint. To lift only t_POLMOD s components, use liftpol. Finally, centerlift allows to lift t_INTMOD s and t_PADIC s using centered residues (lift of smallest absolute value).

liftall(x)

Recursively lift all components of x from $\mathbb{Z}/n\mathbb{Z}$ to \mathbb{Z} , from \mathbb{Q}_p to \mathbb{Q} (as truncate), and polmods to polynomials. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftall, L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

? liftall(x * (1 + 0(3)) + Mod(2,3))
%1 = x + 2
? liftall(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2))
%2 = y*x + 2*z

liftint(x)

Recursively lift all components of x from $\mathbb{Z}/n\mathbb{Z}$ to \mathbb{Z} and from \mathbb{Q}_p to \mathbb{Q} (as truncate). t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftint,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

? liftint(x * (1 + 0(3)) + Mod(2,3))
%1 = x + 2
? liftint(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2))
%2 = Mod(y, y^2 + 1)*x + Mod(Mod(2*z, z^2), y^2 + 1)

liftpol(x)

Recursively lift all components of x which are polmods to polynomials. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftpol,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

? liftpol(x * (1 + 0(3)) + Mod(2,3)) %1 = (1 + 0(3))*x + Mod(2, 3) ? liftpol(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2)) %2 = y*x + Mod(2, 3)*z

limitnum(expr, alpha, precision)

Lagrange-Zagier numerical extrapolation of *expr*, corresponding to a sequence u_n , either given by a closure n - u(n). I.e., assuming that u_n tends to a finite limit ℓ , try to determine ℓ .

The routine assume that u_n has an asymptotic expansion in $n^{-\alpha}$:

$$u_n = \ell + \sum_{i>=1} a_i n^{-i\alpha}$$

for some a_i . It is purely numerical and heuristic, thus may or may not work on your examples. The expression will be evaluated for n = 1, 2, ..., N for an N = O(B) at a bit accuracy bounded by 1.612B.

It is not mandatory to specify α when the u_n have an asymptotic expansion in n^{-1} . However, if the series in n^{-1} is lacunary, specifying α allows faster computation:

```
? \p1000
? limitnum(n->(1+1/n^2)^(n^2)) - exp(1)
time = 1min, 44,681 ms.
%4 = 0.E-1001
? limitnum(n->(1+1/n^2)^(n^2), 2) - exp(1)
time = 27,271 ms.
%5 = 0.E-1001 \\ still perfect, 4 times faster
```

When u_n has an asymptotic expansion in $n^{-\alpha}$ with α not an integer, leaving α unspecified will bring an inexact limit. Giving a satisfying optional argument improves precision; the program runs faster when the optional argument gives non lacunary series.

? \p50 ? limitnum(n->(1+1/n^(7/2))^(n^(7/2))) - exp(1) time = 982 ms. %6 = 4.13[...] E-12 ? limitnum(n->(1+1/n^(7/2))^(n^(7/2)), 1/2) - exp(1) time = 16,745 ms. %7 = 0.E-57 ? limitnum(n->(1+1/n^(7/2))^(n^(7/2)), 7/2) - exp(1) time = 105 ms. %8 = 0.E-57 Alternatively, u_n may be given by a closure $N : -- > [u_1, ..., u_N]$ which can often be programmed in a more efficient way, for instance when u_{n+1} is a simple function of the preceding terms:

```
? \p2000
? limitnum(n -> 2^(4*n+1)*(n!)^4 / (2*n)! /(2*n+1)! ) - Pi
time = 1,755 ms.
%9 = 0.E-2003
? vu(N) = \\ exploit hypergeometric property
{ my(v = vector(N)); v[1] = 8./3;\
for (n=2, N, my(q = 4*n^2); v[n] = v[n-1]*q/(q-1));\
return(v);
}
? limitnum(vu) - Pi \\ much faster
time = 106 ms.
%11 = 0.E-2003
```

All sums and recursions can be handled in the same way. In the above it is essential that u_n be defined as a closure because it must be evaluated at a higher precision than the one expected for the limit. Make sure that the closure does not depend on a global variable which would be computed at a priori fixed accuracy. For instance, precomputing v1 = 8.0/3 first and using v1 in vu above would be wrong because the resulting vector of values will use the accuracy of v1 instead of the ambient accuracy at which limitnum will call it.

Alternatively, and more clumsily, u_n may be given by a vector of values: it must be long and precise enough for the extrapolation to make sense. Let B be the current realbitprecision, the vector length must be at least 1.102B and the values computed with bit accuracy 1.612B.

```
? limitnum(vector(10,n,(1+1/n)^n))
 *** limitnum: nonexistent component in limitnum: index < 43
\setminus at this accuracy, we must have at least 43 values
? limitnum(vector(43,n,(1+1/n)^n)) - exp(1)
\%12 = 0.E - 37
? v = vector(43);
? s = 0; for(i=1,#v, s += 1/i; v[i]= s - log(i));
? limitnum(v) - Euler
%15 = -1.57[...] E-16
? v = vector(43);
\\ ~ 128 bit * 1.612
? localbitprec(207);\
s = 0; for(i=1,#v, s += 1/i; v[i]= s - log(i));
? limitnum(v) - Euler
\%18 = 0.E - 38
```

Because of the above problems, the preferred format is thus a closure, given either a single value or the vector of values $[u_1, ..., u_N]$. The function distinguishes between the two formats by evaluating the closure at N! = 1 and 1 and checking whether it yields vectors of respective length N and 1 or not.

Warning. The expression is evaluated for n = 1, 2, ..., N for an N = O(B) if the current bit accuracy is B. If it is not defined for one of these values, translate or rescale accordingly:

```
? limitnum(n->log(1-1/n)) \\ can't evaluate at n = 1 !
    *** at top-level: limitnum(n->log(1-1/n))
    *** ^-----
```

```
*** in function limitnum: log(1-1/n)
*** ^-----
*** log: domain error in log: argument = 0
? limitnum(n->-log(1-1/(2*n)))
%19 = -6.11[...] E-58
```

We conclude with a complicated example. Since the function is heuristic, it is advisable to check whether it produces the same limit for $u_n, u_{2n}, ... u_{km}$ for a suitable small multiplier k. The following function implements the recursion for the Motzkin numbers M_n which count the number of ways to draw non intersecting chords between n points on a circle:

$$M_n = M_{n-1} + \sum_{i < n-1} M_i M_{n-2-i} = ((n+1)M_{n-1} + (3n-3)M_{n-2})/(n+2).$$

It is known that $M_n (3^{n+1})/(\sqrt{12\pi n^3})$.

```
\ [M_k, M_{k*2}, ..., M_{k*N}] / (3^n / n^(3/2))
vM(N, k = 1) =
\{ my(q = k*N, V); \}
if (q == 1, return ([1/3]));
V = vector(q); V[1] = V[2] = 1;
 for(n = 2, q - 1,
V[n+1] = ((2*n + 1)*V[n] + 3*(n - 1)*V[n-1]) / (n + 2));
 f = (n \rightarrow 3^n / n^{(3/2)});
return (vector(N, n, V[n*k] / f(n*k));
}
? limitnum(vM) - 3/sqrt(12*Pi) \\ complete junk
%1 = 35540390.753542730306762369615276452646
? limitnum(N->vM(N,5)) - 3/sqrt(12*Pi) \\ M_{5n}: better
%2 = 4.130710262178469860 E-25
? limitnum(N->vM(N,10)) - 3/sqrt(12*Pi) \\ M_{10n}: perfect
\%3 = 0.E - 38
? \p2000
? limitnum(N->vM(N,10)) - 3/sqrt(12*Pi) \\ also at high accuracy
time = 409 \text{ ms}.
%4 = 1.1048895470044788191 E-2004
```

In difficult cases such as the above a multiplier of 5 to 10 is usually sufficient. The above example is typical: a good multiplier usually remains sufficient when the requested precision increases!

lindep(v, flag)

finds a small nontrivial integral linear combination between components of v. If none can be found return an empty vector.

If v is a vector with real/complex entries we use a floating point (variable precision) LLL algorithm. If flag = 0 the accuracy is chosen internally using a crude heuristic. If flag > 0 the computation is done with an accuracy of flag decimal digits. To get meaningful results in the latter case, the parameter flag should be smaller than the number of correct decimal digits in the input.

```
? lindep([sqrt(2), sqrt(3), sqrt(2)+sqrt(3)])
%1 = [-1, -1, 1]~
```

If v is p-adic, flag is ignored and the algorithm LLL-reduces a suitable (dual) lattice.

? lindep([1, 2 + 3 + 3² + 3³ + 3⁴ + 0(3⁵)]) %2 = [1, -2]~

If v is a matrix (or a vector of column vectors, or a vector of row vectors), flag is ignored and the function returns a non trivial kernel vector if one exists, else an empty vector.

? lindep([1,2,3;4,5,6;7,8,9])
%3 = [1, -2, 1]~
? lindep([[1,0], [2,0]])
%4 = [2, -1]~
? lindep([[1,0], [0,1]])
%5 = []~

If v contains polynomials or power series over some base field, finds a linear relation with coefficients in the field.

? lindep([x*y, x^2 + y, x^2*y + x*y^2, 1]) %4 = [y, y, -1, -y^2]~

For better control, it is preferable to use t_POL rather than t_SER in the input, otherwise one gets a linear combination which is *t*-adically small, but not necessarily 0. Indeed, power series are first converted to the minimal absolute accuracy occurring among the entries of v (which can cause some coefficients to be ignored), then truncated to polynomials:

? v = [t²+0(t⁴), 1+0(t²)]; L=lindep(v) %1 = [1, 0]~ ? v^{*}L %2 = t²+0(t⁴) \\ small but not 0

listinsert(x, n, _arg3)

Inserts the object x at position n in L (which must be of type t_LIST). This has complexity O(#L - n + 1): all the remaining elements of *list* (from position n + 1 onwards) are shifted to the right. If n is greater than the list length, appends x.

```
? L = List([1,2,3]);
? listput(~L, 4); L \\ listput inserts at end
%4 = List([1, 2, 3, 4])
? listinsert(~L, 5, 1); L \\insert at position 1
%5 = List([5, 1, 2, 3, 4])
? listinsert(~L, 6, 1000); L \\ trying to insert beyond position #L
%6 = List([5, 1, 2, 3, 4, 6]) \\ ... inserts at the end
```

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

listkill(_arg1)

Obsolete, retained for backward compatibility. Just use L = List() instead of listkill(L). In most cases, you won't even need that, e.g. local variables are automatically cleared when a user function returns.

```
listpop(n, _arg2)
```

Removes the *n*-th element of the list *list* (which must be of type t_LIST). If *n* is omitted, or greater than the list current length, removes the last element. If the list is already empty, do nothing. This runs in time O(#L-n+1).

```
? L = List([1,2,3,4]);
? listpop(~L); L \\ remove last entry
%2 = List([1, 2, 3])
```

? listpop(~L, 1); L \\ remove first entry %3 = List([2, 3])

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

 $listput(x, n, _arg3)$

Sets the *n*-th element of the list *list* (which must be of type t_LIST) equal to x. If n is omitted, or greater than the list length, appends x. The function returns the inserted element.

? L = List(); ? listput(~L, 1) %2 = 1 ? listput(~L, 2) %3 = 2 ? L %4 = List([1, 2])

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

You may put an element into an occupied cell (not changing the list length), but it is easier to use the standard list[n] = x construct.

```
? listput(~L, 3, 1) \\ insert at position 1
%5 = 3
? L
%6 = List([3, 2])
? L[2] = 4 \\ simpler
%7 = List([3, 4])
? L[10] = 1 \\ can't insert beyond the end of the list
   *** at top-level: L[10]=1
   *** ^-----
   *** nonexistent component: index > 2
? listput(L, 1, 10) \\ but listput can
%8 = 1
? L
%9 = List([3, 2, 1])
```

This function runs in time O(#L) in the worst case (when the list must be reallocated), but in time O(1) on average: any number of successive listput s run in time O(#L), where #L denotes the list *final* length.

listsort(flag, _arg2)

Sorts the t_LIST *list* in place, with respect to the (somewhat arbitrary) universal comparison function cmp. In particular, the ordering is the same as for sets and setsearch can be used on a sorted list. No value is returned. If flag is nonzero, suppresses all repeated coefficients.

```
? L = List([1,2,4,1,3,-1]); listsort(~L); L
%1 = List([-1, 1, 1, 2, 3, 4])
? setsearch(L, 4)
%2 = 6
? setsearch(L, -2)
%3 = 0
? listsort(~L, 1); L \\ remove duplicates
%4 = List([-1, 1, 2, 3, 4])
```

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place: this is faster than the vecsort command since the list is sorted in place and we avoid unnecessary copies.

```
? v = vector(100,i,random); L = List(v);
? for(i=1,10^4, vecsort(v))
time = 162 ms.
? for(i=1,10^4, vecsort(L))
time = 162 ms.
? for(i=1,10^4, listsort(~L))
time = 63 ms.
```

lngamma(*x*, *precision*)

Principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with nonpositive integers removed, and can have much larger arguments than gamma itself.

For x a power series such that x(0) is not a pole of gamma, compute the Taylor expansion. (PARI only knows about regular power series and can't include logarithmic terms.)

localbitprec(p)

Set the real precision to p bits in the dynamic scope. All computations are performed as if realbitprecision was p: transcendental constants (e.g. Pi) and conversions from exact to floating point inexact data use p bits, as well as iterative routines implicitly using a floating point accuracy as a termination criterion (e.g. solve or intnum). But realbitprecision itself is unaffected and is "unmasked" when we exit the dynamic (*not* lexical) scope. In effect, this is similar to

```
my(bit = default(realbitprecision));
default(realbitprecision,p);
....
default(realbitprecision, bit);
```

but is both less cumbersome, cleaner (no need to manipulate a global variable, which in fact never changes and is only temporarily masked) and more robust: if the above computation is interrupted or an exception occurs, realbitprecision will not be restored as intended.

Such localbitprec statements can be nested, the innermost one taking precedence as expected. Beware that localbitprec follows the semantic of local, not my: a subroutine called from localbitprec scope uses the local accuracy:

```
? f()=bitprecision(1.0);
? f()
%2 = 128
? localbitprec(1000); f()
%3 = 1024
```

Note that the bit precision of *data* (1.0 in the above example) increases by steps of 64 (32 on a 32-bit machine) so we get 1024 instead of the expected 1000; localbitprec bounds the relative error exactly as specified in functions that support that granularity (e.g. lfun), and rounded to the next multiple of 64 (resp. 32) everywhere else.

Warning. Changing realbitprecision or realprecision in programs is deprecated in favor of localbitprec and localprec. Think about the realprecision and realbitprecision defaults as interactive commands for the gp interpreter, best left out of GP programs. Indeed, the above rules imply that mixing both constructs yields surprising results:

```
? \p38
? localprec(19); default(realprecision,1000); Pi
%1 = 3.141592653589793239
? \p
realprecision = 1001 significant digits (1000 digits displayed)
```

Indeed, realprecision itself is ignored within localprec scope, so Pi is computed to a low accuracy. And when we leave the localprec scope, realprecision only regains precedence, it is not "restored" to the original value.

localprec(p)

Set the real precision to p in the dynamic scope and return p. All computations are performed as if realprecision was p: transcendental constants (e.g. Pi) and conversions from exact to floating point inexact data use p decimal digits, as well as iterative routines implicitly using a floating point accuracy as a termination criterion (e.g. solve or intnum). But realprecision itself is unaffected and is "unmasked" when we exit the dynamic (*not* lexical) scope. In effect, this is similar to

```
my(prec = default(realprecision));
default(realprecision,p);
...
default(realprecision, prec);
```

but is both less cumbersome, cleaner (no need to manipulate a global variable, which in fact never changes and is only temporarily masked) and more robust: if the above computation is interrupted or an exception occurs, realprecision will not be restored as intended.

Such localprec statements can be nested, the innermost one taking precedence as expected. Beware that localprec follows the semantic of local, not my: a subroutine called from localprec scope uses the local accuracy:

```
? f()=precision(1.);
? f()
%2 = 38
? localprec(19); f()
%3 = 19
```

Warning. Changing realprecision itself in programs is now deprecated in favor of localprec. Think about the realprecision default as an interactive command for the gp interpreter, best left out of GP programs. Indeed, the above rules imply that mixing both constructs yields surprising results:

```
? \p38
? localprec(19); default(realprecision,100); Pi
%1 = 3.141592653589793239
? \p
realprecision = 115 significant digits (100 digits displayed)
```

Indeed, realprecision itself is ignored within localprec scope, so Pi is computed to a low accuracy. And when we leave localprec scope, realprecision only regains precedence, it is not "restored" to the original value.

log(x, precision)

Principal branch of the natural logarithm of $x \in \mathbb{C}^*$, i.e. such that $\Im(\log(x)) \in]-\pi, \pi]$. The branch cut lies along the negative real axis, continuous with quadrant 2, i.e. such that $\lim_{b\to 0^+} \log(a+bi) = \log a$ for $a \in \mathbb{R}^*$. The result is complex (with imaginary part equal to π) if $x \in \mathbb{R}$ and x < 0. In general, the algorithm uses the formula

$$\log(x) (\pi)/(2agm(1, 4/s)) - m \log 2$$

if $s = x2^m$ is large enough. (The result is exact to B bits provided $s > 2^{B/2}$.) At low accuracies, the series expansion near 1 is used.

p-adic arguments are also accepted for *x*, with the convention that $\log(p) = 0$. Hence in particular $\exp(\log(x))/x$ is not in general equal to 1 but to a (p-1)-th root of unity (or 1 if p = 2) times a power of *p*.

log1p(*x*, *precision*)

Return $\log(1+x)$, computed in a way that is also accurate when the real part of x is near 0. This is the reciprocal function of expm $1(x) = \exp(x) - 1$.

```
? default(realprecision, 10000); x = Pi*1e-100;
? (expm1(log1p(x)) - x) / x
%2 = -7.668242895059371866 E-10019
? (log1p(expm1(x)) - x) / x
%3 = -7.668242895059371866 E-10019
```

When x is small, this function is both faster and more accurate than log(1 + x):

```
? \p38
? x = 1e-20;
? localprec(100); c = log1p(x); \\ reference point
? a = log1p(x); abs((a - c)/c)
%6 = 0.E-38
? b = log(1+x); abs((b - c)/c) \\ slightly less accurate
%7 = 1.5930919111324522770 E-38
? for (i=1,10^5,log1p(x))
time = 81 ms.
? for (i=1,10^5,log(1+x))
time = 100 ms. \\ slower, too
```

logint(x, b, z)

Return the largest integer e so that $b^e \le x$, where the parameters b > 1 and x > 0 are both integers. If the parameter z is present, set it to b^e .

? logint(1000, 2) %1 = 9 ? 2^9 %2 = 512 ? logint(1000, 2, &z) %3 = 9 ? z %4 = 512

The number of digits used to write b in base x is 1 + logint(x,b):

? #digits(1000!, 10)
%5 = 2568
? logint(1000!, 10)
%6 = 2567

This function may conveniently replace

floor(log(x) / log(b))

which may not give the correct answer since PARI does not guarantee exact rounding.

mapdelete(x, _arg2)

Removes x from the domain of the map M.

? M = Map(["a",1; "b",3; "c",7]); ? mapdelete(M,"b"); ? Mat(M) ["a" 1] ["c" 7]

mapget(M, x)

Returns the image of x by the map M.

```
? M=Map(["a",23;"b",43]);
? mapget(M,"a")
%2 = 23
? mapget(M,"b")
%3 = 43
```

Raises an exception when the key x is not present in M.

```
? mapget(M,"c")
 *** at top-level: mapget(M,"c")
 *** ^_____
*** mapget: nonexistent component in mapget: index not in map
```

mapisdefined(M, x, z)

Returns true (1) if x has an image by the map M, false (0) otherwise. If z is present, set z to the image of x, if it exists.

? M1 = Map([1, 10; 2, 20]); ? mapisdefined(M1,3) %1 = 0 ? mapisdefined(M1, 1, &z) %2 = 1 ? z %3 = 10

? M2 = Map(); N = 19; ? for (a=0, N-1, mapput(M2, a^3%N, a)); ? {for (a=0, N-1, if (mapisdefined(M2, a, &b),

```
printf("%d is the cube of %d mod %d\n",a,b,N)));}
0 is the cube of 0 mod 19
1 is the cube of 11 mod 19
7 is the cube of 9 mod 19
8 is the cube of 14 mod 19
11 is the cube of 17 mod 19
12 is the cube of 15 mod 19
18 is the cube of 18 mod 19
```

mapput(*x*, *y*, _*arg3*)

Associates x to y in the map M. The value y can be retrieved with mapget.

```
? M = Map();
? mapput(~M, "foo", 23);
? mapput(~M, 7718, "bill");
? mapget(M, "foo")
%4 = 23
? mapget(M, 7718)
%5 = "bill"
? Vec(M) \\ keys
%6 = [7718, "foo"]
? Mat(M)
%7 =
[ 7718 "bill"]
["foo" 23]
```

matadjoint(M, flag)

adjoint matrix of M, i.e. a matrix N of cofactors of M, satisfying $M * N = \det(M) * \operatorname{Id}$. M must be a (not necessarily invertible) square matrix of dimension n. If flag is 0 or omitted, we try to use Leverrier-Faddeev's algorithm, which assumes that n! invertible. If it fails or flag = 1, compute T = charpoly(M) independently first and return $(-1)^{n-1}(T(x) - T(0))/x$ evaluated at M.

? a = [1,2,3;3,4,5;6,7,8] * Mod(1,4); ? matadjoint(a) %2 = [Mod(1, 4) Mod(1, 4) Mod(2, 4)] [Mod(2, 4) Mod(2, 4) Mod(0, 4)] [Mod(1, 4) Mod(1, 4) Mod(2, 4)]

Both algorithms use $O(n^4)$ operations in the base ring. Over a field, they are usually slower than computing the characteristic polynomial or the inverse of M directly.

matalgtobasis(nf, x)

This function is deprecated, use apply.

nf being a number field in nfinit format, and x a (row or column) vector or matrix, apply nfalgtobasis to each entry of x.

matbasistoalg(nf, x)

This function is deprecated, use apply.

nf being a number field in nfinit format, and x a (row or column) vector or matrix, apply nfbasistoalg to each entry of x.

matcompanion(x)

The left companion matrix to the nonzero polynomial x.

matconcat(v)

Returns a t_MAT built from the entries of v, which may be a t_VEC (concatenate horizontally), a t_COL (concatenate vertically), or a t_MAT (concatenate vertically each column, and concatenate vertically the resulting matrices). The entries of v are always considered as matrices: they can themselves be t_VEC (seen as a row matrix), a t_COL seen as a column matrix), a t_MAT, or a scalar (seen as an 1x1 matrix).

```
? A=[1,2;3,4]; B=[5,6]~; C=[7,8]; D=9;
? matconcat([A, B]) \\ horizontal
%1 =
[1 2 5]
[3 4 6]
? matconcat([A, C]~) \\ vertical
%2 =
[1 2]
[3 4]
[7 8]
? matconcat([A, B; C, D]) \\ block matrix
%3 =
[1 2 5]
[3 4 6]
[7 8 9]
```

If the dimensions of the entries to concatenate do not match up, the above rules are extended as follows:

- each entry $v_{i,j}$ of v has a natural length and height: 1x1 for a scalar, 1xn for a t_VEC of length n, nx1 for a t_COL, mxn for an mxn t_MAT
- let H_i be the maximum over j of the lengths of the $v_{i,j}$, let L_j be the maximum over i of the heights of the $v_{i,j}$. The dimensions of the (i, j)-th block in the concatenated matrix are $H_i x L_j$.
- a scalar $s = v_{i,j}$ is considered as s times an identity matrix of the block dimension $\min(H_i, L_j)$
- blocks are extended by 0 columns on the right and 0 rows at the bottom, as needed.

```
? matconcat([1, [2,3]~, [4,5,6]~]) \\ horizontal
%4 =
[1 2 4]
[0 3 5]
[0 0 6]
? matconcat([1, [2,3], [4,5,6]]~) \\ vertical
%5 =
[1 0 0]
```

```
[2 3 0]
[4 5 6]
? matconcat([B, C; A, D]) \\ block matrix
%6 =
[5 0 7 8]
[6 0 0 0]
[1 2 9 0]
[3 4 0 9]
? U=[1,2;3,4]; V=[1,2,3;4,5,6;7,8,9];
? matconcat(matdiagonal([U, V])) \\ block diagonal
%7 =
[1 2 0 0 0]
[34000]
[0 0 1 2 3]
[0 0 4 5 6]
[0 0 7 8 9]
```

matdet(x, flag)

Determinant of the square matrix x.

If flag = 0, uses an appropriate algorithm depending on the coefficients:

- integer entries: modular method due to Dixon, Pernet and Stein.
- real or *p*-adic entries: classical Gaussian elimination using maximal pivot.
- intmod entries: classical Gaussian elimination using first nonzero pivot.
- other cases: Gauss-Bareiss.

If flag = 1, uses classical Gaussian elimination with appropriate pivoting strategy (maximal pivot for real or *p*-adic coefficients). This is usually worse than the default.

matdetint(B)

Let B be an mxn matrix with integer coefficients. The *determinant* D of the lattice generated by the columns of B is the square root of det (B^TB) if B has maximal rank m, and 0 otherwise.

This function uses the Gauss-Bareiss algorithm to compute a positive *multiple* of D. When B is square, the function actually returns $D = \| \det B \|$.

This function is useful in conjunction with mathnfmod, which needs to know such a multiple. If the rank is maximal but the matrix is nonsquare, you can obtain D exactly using

matdet(mathnfmod(B, matdetint(B)))

Note that as soon as one of the dimensions gets large (m or n is larger than 20, say), it will often be much faster to use mathnf(B, 1) or mathnf(B, 4) directly.

matdetmod(x, d)

Given a matrix x with t_INT entries and d an arbitrary positive integer, return the determinant of x modulo d.

```
? A = [4,2,3; 4,5,6; 7,8,9]
? matdetmod(A,27)
%2 = 9
```

Note that using the generic function matdet on a matrix with t_INTMOD entries uses Gaussian reduction and will fail in general when the modulus is not prime.

```
? matdet(A * Mod(1,27))
 *** at top-level: matdet(A*Mod(1,27))
 *** ^_____
*** matdet: impossible inverse in Fl_inv: Mod(3, 27).
```

matdiagonal(x)

x being a vector, creates the diagonal matrix whose diagonal entries are those of x.

```
? matdiagonal([1,2,3]);
%1 =
[1 0 0]
[0 2 0]
[0 0 3]
```

Block diagonal matrices are easily created using matconcat:

```
? U=[1,2;3,4]; V=[1,2,3;4,5,6;7,8,9];
? matconcat(matdiagonal([U, V]))
%1 =
[1 2 0 0 0]
[3 4 0 0 0]
[0 0 1 2 3]
[0 0 4 5 6]
[0 0 7 8 9]
```

mateigen(x, flag, precision)

Returns the (complex) eigenvectors of x as columns of a matrix. If flag = 1, return [L, H], where L contains the eigenvalues and H the corresponding eigenvectors; multiple eigenvalues are repeated according to the eigenspace dimension (which may be less than the eigenvalue multiplicity in the characteristic polynomial).

This function first computes the characteristic polynomial of x and approximates its complex roots (λ_i) , then tries to compute the eigenspaces as kernels of the $x - \lambda_i$. This algorithm is ill-conditioned and is likely to miss kernel vectors if some roots of the characteristic polynomial are close, in particular if it has multiple roots.

? A = [13,2; 10,14]; mateigen(A) %1 = [-1/2 2/5]

```
[ 1 1]
? [L,H] = mateigen(A, 1);
? L
%3 = [9, 18]
? H
%4 =
[-1/2 2/5]
[ 1 1]
? A * H == H * matdiagonal(L)
%5 = 1
```

For symmetric matrices, use qfjacobi instead; for Hermitian matrices, compute

A = real(x); B = imag(x); y = matconcat([A, -B; B, A]);

and apply qfjacobi to y.

matfrobenius(M, flag, v)

Returns the Frobenius form of the square matrix M. If flag = 1, returns only the elementary divisors as a vector of polynomials in the variable v. If flag = 2, returns a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.

mathess(x)

Returns a matrix similar to the square matrix x, which is in upper Hessenberg form (zero entries below the first subdiagonal).

mathilbert(n)

x being a long, creates the Hilbert matrix of order x, i.e. the matrix whose coefficient (i,:math;j) is 1/(i+j-1).

mathnf(M, flag)

Let R be a Euclidean ring, equal to \mathbb{Z} or to K[X] for some field K. If M is a (not necessarily square) matrix with entries in R, this routine finds the *upper triangular* Hermite normal form of M. If the rank of M is equal to its number of rows, this is a square matrix. In general, the columns of the result form a basis of the R-module spanned by the columns of M.

The values of *flag* are:

- 0 (default): only return the Hermite normal form H
- 1 (complete output): return [H, U], where H is the Hermite normal form of M, and U is a transformation matrix such that MU = [0||H]. The matrix U belongs to GL(R). When M has a large kernel, the entries of U are in general huge.

For these two values, we use a naive algorithm, which behaves well in small dimension only. Larger values correspond to different algorithms, are restricted to *integer* matrices, and all output the unimodular matrix U. From now on all matrices have integral entries.

• flag = 4, returns [H, U] as in "complete output" above, using a variant of LLL reduction along the way. The matrix U is provably small in the L_2 sense, and often close to optimal; but the reduction is in general slow, although provably polynomial-time. If flag = 5, uses Batut's algorithm and output [H, U, P], such that H and U are as before and P is a permutation of the rows such that P applied to MU gives H. This is in general faster than flag = 4 but the matrix U is usually worse; it is heuristically smaller than with the default algorithm.

When the matrix is dense and the dimension is large (bigger than 100, say), flag = 4 will be fastest. When M has maximal rank, then

H = mathnfmod(M, matdetint(M))

will be even faster. You can then recover U as $M^{-1}H$.

```
? M = matrix(3,4,i,j,random([-5,5]))
%1 =
[0230]
[-5 3 -5 -5]
[43-54]
? [H,U] = mathnf(M, 1);
? U
%3 =
[-1 \ 0 \ -1 \ 0]
[ 0 5 3 2]
[ 0 3 1 1]
[1000]
? H
%5 =
[19 9 7]
[ 0 9 1]
[001]
? M*U
%6 =
[0 19 9 7]
[0 0 9 1]
[0 0 0 1]
```

For convenience, M is allowed to be a t_VEC, which is then automatically converted to a t_MAT, as per the Mat function. For instance to solve the generalized extended gcd problem, one may use

```
? v = [116085838, 181081878, 314252913,10346840];
? [H,U] = mathnf(v, 1);
? U
%2 =
[ 103 -603 15 -88]
```

[-146 13 -1208 352] [58 220 678 -167] [-362 -144 381 -101] ? v*U %3 = [0, 0, 0, 1]

This also allows to input a matrix as a t_VEC of t_COL s of the same length (which Mat would concatenate to the t_MAT having those columns):

? v = [[1,0,4]~, [3,3,4]~, [0,-4,-5]~]; mathnf(v) %1 = [47 32 12] [0 1 0] [0 0 1]

mathnfmod(x, d)

If x is a (not necessarily square) matrix of maximal rank with integer entries, and d is a multiple of the (nonzero) determinant of the lattice spanned by the columns of x, finds the *upper triangular* Hermite normal form of x.

If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x. Even when d is known, this is in general slower than mathnf but uses much less memory.

mathnfmodid(x, d)

Outputs the (upper triangular) Hermite normal form of x concatenated with the diagonal matrix with diagonal d. Assumes that x has integer entries. Variant: if d is an integer instead of a vector, concatenate d times the identity matrix.

```
? m=[0,7;-1,0;-1,-1]
%1 =
[ 0 7]
[-1 0]
[-1 -1]
? mathnfmodid(m, [6,2,2])
%2 =
[2 1 1]
[0 1 0]
[0 0 1]
? mathnfmodid(m, 10)
%3 =
[10 7 3]
[ 0 1 0]
```

[001]

mathouseholder(Q, v)

applies a sequence Q of Householder transforms, as returned by matqr(M, 1) to the vector or matrix v.

```
? m = [2,1; 3,2]; \\ some random matrix
? [Q,R] = matqr(m);
?Q
%3 =
[-0.554... -0.832...]
[-0.832... 0.554...]
? R
%4 =
[-3.605... -2.218...]
[0 0.277...]
? v = [1, 2]~; \\ some random vector
? Q * v
%6 = [-2.218..., 0.277...]~
? [q,r] = matqr(m, 1);
? exponent(r - R) \setminus r is the same as R
\%8 = -128
? q \setminus but q has a different structure
%9 = [[0.0494..., [5.605..., 3]]]]
? mathouseholder(q, v) \setminus applied to v
%10 = [-2.218..., 0.277...]~
```

The point of the Householder structure is that it efficiently represents the linear operator v : -- > Qv in a more stable way than expanding the matrix Q:

```
? m = mathilbert(20); v = vectorv(20,i,i^2+1);
? [Q,R] = matqr(m);
? [q,r] = matqr(m, 1);
? \p100
? [q2,r2] = matqr(m, 1); \\ recompute at higher accuracy
? exponent(R - r)
%5 = -127
? exponent(R - r2)
%6 = -127
? exponent(mathouseholder(q,v) - mathouseholder(q2,v))
%7 = -119
? exponent(Q*v - mathouseholder(q2,v))
%8 = 9
```

We see that R is OK with or without a flag to mature but that multiplying by Q is considerably less precise than applying the sequence of Householder transforms encoded by q.

matid(n)

Creates the nxn identity matrix.

matimage(x, flag)

Gives a basis for the image of the matrix x as columns of a matrix. A priori the matrix can have entries of any type. If flag = 0, use standard Gauss pivot. If flag = 1, use matsupplement (much slower: keep the default flag!).

matimagecompl(x)

Gives the vector of the column indices which are not extracted by the function matimage, as a permutation (t_VECSMALL). Hence the number of components of matimagecompl(x) plus the number of columns of matimage(x) is equal to the number of columns of the matrix x.

matimagemod(x, d, U)

Gives a Howell basis (unique representation for submodules of $(\mathbb{Z}/d\mathbb{Z})^n$) for the image of the matrix x modulo d as columns of a matrix H. The matrix x must have t_INT entries, and d can be an arbitrary positive integer. If U is present, set it to a matrix such that AU = H.

```
? A = [2,1;0,2];
? matimagemod(A,6,&U)
%2 =
[1 0]
[0 2]
? U
%3 =
[5 1]
[3 4]
? (A*U)%6
%4 =
[1 0]
[0 2]
```

Caveat. In general the number of columns of the Howell form is not the minimal number of generators of the submodule. Example:

? matimagemod([1;2],4)
%5 =
[2 1]
[0 2]

Caveat 2. In general the matrix U is not invertible, even if A and H have the same size. Example:

```
? matimagemod([4,1;0,4],8,&U)
%6 =
[2 1]
[0 4]
? U
%7 =
[0 0]
```

[2 1]			

matindexrank(M)

M being a matrix of rank r, returns a vector with two t_VECSMALL components y and z of length r giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these two vectors using vecextract(M, y, z) is invertible. The vectors y and z are sorted in increasing order.

matintersect(x, y)

x and y being two matrices with the same number of rows each of whose columns are independent, finds a basis of the vector space equal to the intersection of the spaces spanned by the columns of x and y respectively. The faster function idealintersect can be used to intersect fractional ideals (projective \mathbb{Z}_K modules of rank 1); the slower but more general function nfhnf can be used to intersect general \mathbb{Z}_K -modules.

matinverseimage(x, y)

Given a matrix x and a column vector or matrix y, returns a preimage z of y by x if one exists (i.e such that xz = y), an empty vector or matrix otherwise. The complete inverse image is z + Kerx, where a basis of the kernel of x may be obtained by matker.

```
? M = [1,2;2,4];
? matinverseimage(M, [1,2]~)
%2 = [1, 0]~
? matinverseimage(M, [3,4]~)
%3 = []~ \\ no solution
? matinverseimage(M, [1,3,6;2,6,12])
%4 =
[1 3 6]
[0 0 0]
? matinverseimage(M, [1,2;3,4])
%5 = [;] \\ no solution
? K = matker(M)
%6 =
[-2]
[1]
```

matinvmod(x, d)

Computes a left inverse of the matrix x modulo d. The matrix x must have t_INT entries, and d can be an arbitrary positive integer.

? A = [3,1,2;1,2,1;3,1,1]; ? U = matinvmod(A,6) %2 = [1 1 3] [2 3 5] [1 0 5] ? (U*A)%6 %3 =

[1 0 0] [0 1 0] [0 0 1] ? matinvmod(A,5) *** at top-level: matinvmod(A,5) *** ^_____ *** matinvmod: impossible inverse in gen_inv: 0.

matisdiagonal(x)

Returns true (1) if x is a diagonal matrix, false (0) if not.

matker(x, flag)

Gives a basis for the kernel of the matrix x as columns of a matrix. The matrix can have entries of any type, provided they are compatible with the generic arithmetic operations (+, x and /).

If x is known to have integral entries, set flag = 1.

matkerint(x, flag)

Gives an LLL-reduced \mathbb{Z} -basis for the lattice equal to the kernel of the matrix x with rational entries. *flag* is deprecated, kept for backward compatibility.

matkermod(x, d, im)

Gives a Howell basis (unique representation for submodules of $(\mathbb{Z}/d\mathbb{Z})^n$, cf. matimagemod) for the kernel of the matrix x modulo d as columns of a matrix. The matrix x must have t_INT entries, and d can be an arbitrary positive integer. If im is present, set it to a basis of the image of x (which is computed on the way).

```
? A = [1,2,3;5,1,4]
%1 =
[1 2 3]
[5 1 4]
? K = matkermod(A,6)
%2 =
[2 1]
[2 1]
[0 3]
? (A*K)%6
%3 =
[0 0]
[0 0]
```

matmuldiagonal(x, d)

Product of the matrix x by the diagonal matrix whose diagonal entries are those of the vector d. Equivalent to, but much faster than x * matdiagonal(d).

matmultodiagonal(x, y)

Product of the matrices x and y assuming that the result is a diagonal matrix. Much faster than x * y in that case.
The result is undefined if x * y is not diagonal.

matpascal(n, q)

Creates as a matrix the lower triangular Pascal triangle of order x + 1 (i.e. with binomial coefficients up to x). If q is given, compute the q-Pascal triangle (i.e. using q-binomial coefficients).

matpermanent(x)

Permanent of the square matrix x using Ryser's formula in Gray code order.

? n = 20; m = matrix(n,n,i,j, i!=j); ? matpermanent(m) %2 = 895014631192902121 ? n! * sum(i=0,n, (-1)^i/i!) %3 = 895014631192902121

This function runs in time $O(2^n n)$ for a matrix of size n and is not implemented for n large.

matqr(M, flag, precision)

Returns [Q, R], the QR-decomposition of the square invertible matrix M with real entries: Q is orthogonal and R upper triangular. If flag = 1, the orthogonal matrix is returned as a sequence of Householder transforms: applying such a sequence is stabler and faster than multiplication by the corresponding Q matrix. More precisely, if

[Q,R] = matqr(M); [q,r] = matqr(M, 1);

then r = R and mathouseholder(q, M) is (close to) R; furthermore

```
mathouseholder(q, matid(#M)) == Q~
```

the inverse of Q. This function raises an error if the precision is too low or x is singular.

matrank(x)

Rank of the matrix x.

matreduce(m)

Let m be a factorization matrix, i.e., a 2-column matrix whose columns contains arbitrary "generators" and integer "exponents" respectively. Returns the canonical form of m: the first column is sorted with unique elements and the second one contains the merged "exponents" (exponents of identical entries in the first column of m are added, rows attached to 0 exponents are deleted). The generators are sorted with respect to the universal cmp routine; in particular, this function is the identity on true integer factorization matrices, but not on other factorizations (in products of polynomials or maximal ideals, say). It is idempotent.

For convenience, this function also allows a vector m, which is handled as a factorization with all exponents equal to 1, as in factorback.

? A=[x,2;y,4]; B=[x,-2; y,3; 3, 4]; C=matconcat([A,B]~)
%1 =
[x 2]
[y 4]
[x -2]
[y 3]

[3 4]
? matreduce(C)
%2 =
[3 4]
[y 7]
? matreduce([x,x,y,x,z,x,y]) \\ vector argument
%3 =
[x 4]
[y 2]
[z 1]

matrixqz(A, p)

A being an mxn matrix in $M_{m,n}(\mathbb{Q})$, let $Im_{\mathbb{Q}}A$ (resp. $Im_{\mathbb{Z}}A$) the \mathbb{Q} -vector space (resp. the \mathbb{Z} -module) spanned by the columns of A. This function has varying behavior depending on the sign of p:

If $p \ge 0$, A is assumed to have maximal rank $n \le m$. The function returns a matrix $B \in M_{m,n}(\mathbb{Z})$, with $Im_{\mathbb{Q}}B = Im_{\mathbb{Q}}A$, such that the GCD of all its nxn minors is coprime to p; in particular, if p = 0 (default), this GCD is 1.

If p = -1, returns a basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{Z}}A$.

If p = -2, returns a basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{Q}}A$.

Caveat. (p = -1 or -2) For efficiency reason, we do not compute the HNF of the resulting basis.

```
? minors(x) = vector(#x[,1], i, matdet(x[^i,]));
? A = [3,1/7; 5,3/7; 7,5/7]; minors(A)
%1 = [4/7, 8/7, 4/7] \setminus  determinants of all 2x2 minors
? B = matrixqz(A)
%2 =
[3 1]
[5 2]
[7 3]
? minors(%)
\%3 = [1, 2, 1] \setminus B integral with coprime minors
? matrixqz(A,-1)
%4 =
[3 1]
[5 3]
[7 5]
? matrixqz(A,-2)
%5 =
[3 1]
```

[5 2]			
[7 3]			

matsize(x)

x being a vector or matrix, returns a row vector with two components, the first being the number of rows (1 for a row vector), the second the number of columns (1 for a column vector).

matsnf(X, flag)

If X is a (singular or nonsingular) matrix outputs the vector of elementary divisors of X, i.e. the diagonal of the Smith normal form of X, normalized so that $d_n || d_{n-1} || ... || d_1$. X must have integer or polynomial entries; in the latter case, X must be a square matrix.

The binary digits of *flag* mean:

1 (complete output): if set, outputs [U, V, D], where U and V are two unimodular matrices such that UXV is the diagonal matrix D. Otherwise output only the diagonal of D. If X is not a square matrix, then D will be a square diagonal matrix padded with zeros on the left or the top.

4 (cleanup): if set, cleans up the output. This means that elementary divisors equal to 1 will be deleted, i.e. outputs a shortened vector D' instead of D. If complete output was required, returns [U', V', D'] so that U'XV' = D'holds. If this flag is set, X is allowed to be of the form vector of elementary divisors' or :math: `[U,V,D] as would normally be output with the cleanup flag unset.

matsolve(M, B)

Let M be a left-invertible matrix and B a column vector such that there exists a solution X to the system of linear equations MX = B; return the (unique) solution X. This has the same effect as, but is faster, than $M^{-1} * B$. Uses Dixon p-adic lifting method if M and B are integral and Gaussian elimination otherwise. When there is no solution, the function returns an X such that MX - B is nonzero although it has at least #M zero entries:

```
? M = [1,2;3,4;5,6];
? B = [4,6,8]~; X = matsolve(M, B)
%2 = [-2, 3]~
? M*X == B
%3 = 1
? B = [1,2,4]~; X = matsolve(M, [1,2,4]~)
%4 = [0, 1/2]~
? M*X - B
%5 = [0, 0, -1]~
```

Raises an exception if M is not left-invertible, even if there is a solution:

```
? M = [1,1;1,1]; matsolve(M, [1,1]~)
   *** at top-level: matsolve(M,[1,1]~)
   *** ^------
   *** matsolve: impossible inverse in gauss: [1, 1; 1, 1].
```

The function also works when B is a matrix and we return the unique matrix solution X provided it exists.

matsolvemod(M, D, B, flag)

M being any integral matrix, *D* a column vector of nonnegative integer moduli, and *B* an integral column vector, gives an integer solution to the system of congruences $\sum_{i} m_{i,j} x_j = b_i (modd_i)$ if one exists, otherwise returns zero. Shorthand notation: *B* (resp. *D*) can be given as a single integer, in which case all the b_i (resp. d_i) above are taken to be equal to *B* (resp. *D*).

? M = [1,2;3,4]; ? matsolvemod(M, [3,4]~, [1,2]~) %2 = [10, 0]~ ? matsolvemod(M, 3, 1) \\ M X = [1,1]~ over F_3 %3 = [2, 1]~ ? matsolvemod(M, [3,0]~, [1,2]~) \\ x + 2y = 1 (mod 3), 3x + 4y = 2 (in Z) %4 = [6, -4]~

If flag = 1, all solutions are returned in the form of a two-component row vector [x, u], where x is an integer solution to the system of congruences and u is a matrix whose columns give a basis of the homogeneous system (so that all solutions can be obtained by adding x to any linear combination of columns of u). If no solution exists, returns zero.

matsupplement(x)

Assuming that the columns of the matrix x are linearly independent (if they are not, an error message is issued), finds a square invertible matrix whose first columns are the columns of x, i.e. supplement the columns of x to a basis of the whole space.

? matsupplement([1;2])
%1 =
[1 0]
[2 1]

Raises an error if x has 0 columns, since (due to a long standing design bug), the dimension of the ambient space (the number of rows) is unknown in this case:

mattranspose(x)

Transpose of x (also x). This has an effect only on vectors and matrices.

$\max(x, y)$

Creates the maximum of x and y when they can be compared.

mfDelta()

Mf structure corresponding to the Ramanujan Delta function Δ .

```
? mfcoefs(mfDelta(),4)
%1 = [0, 1, -24, 252, -1472]
```

mfEH(k)

k being in $1/2 + \mathbb{Z}_{>=0}$, return the mf structure corresponding to the Cohen-Eisenstein series H_k of weight k on $\Gamma_0(4)$.

```
? H = mfEH(13/2); mfcoefs(H,4)
%1 = [691/32760, -1/252, 0, 0, -2017/252]
```

The coefficients of H are given by the Cohen-Hurwitz function H(k-1/2, N) and can be obtained for moderately large values of N (the algorithm uses O(N) time):

```
? mfcoef(H,10^5+1)
time = 55 ms.
%2 = -12514802881532791504208348
? mfcoef(H,10^7+1)
time = 6,044 ms.
%3 = -1251433416009877455212672599325104476
```

mfEk(k)

K being an even nonnegative integer, return the mf structure corresponding to the standard Eisenstein series E_k .

```
? mfcoefs(mfEk(8), 4)
%1 = [1, 480, 61920, 1050240, 7926240]
```

mfTheta(psi)

The unary theta function corresponding to the primitive Dirichlet character ψ . Its level is $4F(\psi)^2$ and its weight is $1 - \psi(-1)/2$.

```
? Ser(mfcoefs(mfTheta(),30))
%1 = 1 + 2*x + 2*x^4 + 2*x^9 + 2*x^16 + 2*x^25 + 0(x^31)
? f = mfTheta(8); Ser(mfcoefs(f,30))
%2 = 2*x - 2*x^9 - 2*x^25 + 0(x^31)
? mfparams(f)
%3 = [256, 1/2, 8, y, t + 1]
? g = mfTheta(-8); Ser(mfcoefs(g,30))
%4 = 2*x + 6*x^9 - 10*x^25 + 0(x^31)
? mfparams(g)
%5 = [256, 3/2, 8, y, t + 1]
? h = mfTheta(Mod(2,5)); mfparams(h)
%6 = [100, 3/2, Mod(7, 20), y, t^2 + 1]
```

mfatkin(*mfatk*, *f*)

Given a mfatk output by mfatk = mfatkininit(mf,Q) and a modular form f belonging to the pace mf, returns the modular form $g = Cxf || W_Q$, where C = mfatk[3] is a normalizing constant such that g has the same field of coefficients as f; mfatk[3] gives the constant C, and mfatk[1] gives the modular form space to which gbelongs (or is set to 0 if it is mf).

```
? mf = mfinit([35,2],0); [f] = mfbasis(mf);
? mfcoefs(f, 4)
%2 = [0, 3, -1, 0, 3]
? mfatk = mfatkininit(mf,7);
? g = mfatkin(mfatk, f); mfcoefs(g, 4)
%4 = [0, 1, -1, -2, 7]
? mfatk = mfatkininit(mf,35);
? g = mfatkin(mfatk, f); mfcoefs(g, 4)
%6 = [0, -3, 1, 0, -3]
```

mfatkineigenvalues(mf, Q, precision)

Given a modular form space mf of integral weight k and a primitive divisor Q of the level N of mf, outputs the Atkin-Lehner eigenvalues of w_Q on the new space, grouped by orbit. If the Nebentypus χ of mf is a (trivial or) quadratic character defined modulo N/Q, the result is rounded and the eigenvalues are i^k .

? mf = mfinit([35,2],0); mffields(mf) %1 = [y, y^2 - y - 4] \\ two orbits, dimension 1 and 2 ? mfatkineigenvalues(mf,5) %2 = [[1], [-1, -1]] ? mf = mfinit([12,7,Mod(3,4)],0); ? mfatkineigenvalues(mf,3) %4 = [[I, -I, -I, I, I, -I]] \\ one orbit

To obtain the eigenvalues on a larger space than the new space, e.g., the full space, you can directly call [mfB, M,C] = mfatkininit and compute the eigenvalues as the roots of the characteristic polynomial of M/C, by dividing the roots of charpoly(M) by C. Note that the characteristic polynomial is computed exactly since M has coefficients in $\mathbb{Q}(\chi)$, whereas C may be given by a complex number. If the coefficients of the characteristic polynomial are polmods modulo T they must be embedded to \mathbb{C} first using subst(lift(), t, exp(2*I*Pi/n)), when T is poliscyclo(n); note that T = mf.mod.

mfatkininit(*mf*, *Q*, *precision*)

Given a modular form space with parameters N, k, χ and a primitive divisor Q of the level N, initializes data necessary for working with the Atkin-Lehner operator W_Q , for now only the function mfatkin. We write $\chi \chi_Q \chi_{N/Q}$ where the two characters are primitive with (coprime) conductors dividing Q and N/Q respectively. For $F \in M_k(\Gamma_0(N), \chi)$, the form $F || W_Q$ still has level N and weight k but its Nebentypus may no longer be χ : it becomes $\overline{\chi_Q} \chi_{N/Q}$) if k is integral and $\overline{\chi_Q} \chi_{N/Q})(4Q/.)$ if not.

The result is a technical 4-component vector [mfB, MC, C, mf], where

- mfB encodes the modular form space to which $F || W_Q$ belongs when $F \in M_k(\Gamma_0(N), \chi)$: an mfinit corresponding to a new Nebentypus or the integer 0 when the character does not change. This does not depend on F.
- MC is the matrix of W_Q on the bases of mf and mfB multiplied by a normalizing constant $C(k, \chi, Q)$. This matrix has polmod coefficients in $\mathbb{Q}(\chi)$.
- C is the complex constant $C(k, \chi, Q)$. For k integral, let $A(k, \chi, Q) = Q^{\varepsilon}/g(\chi_Q)$, where $\varepsilon = 0$ for k even and 1/2 for k odd and where $g(\chi_Q)$ is the Gauss sum attached to χ_Q). (A similar, more complicated, definition holds in half-integral weight depending on the parity of k 1/2.) Then if M denotes the matrix of W_Q on the bases of mf and mfB, A.M has coefficients in $\mathbb{Q}(\chi)$. If A is rational, we let C = 1 and C = A as a floating point complex number otherwise, and finally MC := M.C.

```
? mf=mfinit([32,4],0); [mfB,MC,C]=mfatkininit(mf,32); MC
%1 =
[5/16 11/2 55/8]
[ 1/8 0 -5/4]
[1/32 -1/4 11/16]
? C
%2 = 1
? mf=mfinit([32,4,8],0); [mfB,MC,C]=mfatkininit(mf,32); MC
%3 =
[ 1/8 -7/4]
[-1/16 -1/8]
? C
%4 = 0.35355339059327376220042218105242451964
? algdep(C,2) \\ C = 1/sqrt(8)
```

 $85 = 8 \times x^2 - 1$

mfbasis(*NK*, *space*)

If NK = [N, k, CHI] as in mfinit, gives a basis of the corresponding subspace of $M_k(\Gamma_0(N), \chi)$. NK can also be the output of mfinit, in which case space can be omitted. To obtain the eigenforms, use mfeigenbasis.

If space is a full space M_k , the output is the union of first, a basis of the space of Eisenstein series, and second, a basis of the cuspidal space.

```
? see(L) = apply(f->mfcoefs(f,3), L);
? mf = mfinit([35,2],0);
? see( mfbasis(mf) )
%2 = [[0, 3, -1, 0], [0, -1, 9, -8], [0, 0, -8, 10]]
? see( mfeigenbasis(mf) )
%3 = [[0, 1, 0, 1], [Mod(0, z^2 - z - 4), Mod(1, z^2 - z - 4), \
Mod(-z, z^2 - z - 4), Mod(z - 1, z^2 - z - 4)]]
? mf = mfinit([35,2]);
? see( mfbasis(mf) )
%5 = [[1/6, 1, 3, 4], [1/4, 1, 3, 4], [17/12, 1, 3, 4], \
[0, 3, -1, 0], [0, -1, 9, -8], [0, 0, -8, 10]]
? see( mfbasis([48,4],0) )
%6 = [[0, 3, 0, -3], [0, -3, 0, 27], [0, 2, 0, 30]]
```

$\mathbf{mfbd}(F, d)$

F being a generalized modular form, return B(d)(F), where B(d) is the expanding operator $\tau : -- > d\tau$.

? D2=mfbd(mfDelta(),2); mfcoefs(D2, 6) %1 = [0, 0, 1, 0, -24, 0, 252]

mfbracket(F, G, m)

Compute the m-th Rankin-Cohen bracket of the generalized modular forms F and G.

```
? E4 = mfEk(4); E6 = mfEk(6);
? D1 = mfbracket(E4,E4,2); mfcoefs(D1,5)/4800
%2 = [0, 1, -24, 252, -1472, 4830]
? D2 = mfbracket(E4,E6,1); mfcoefs(D2,10)/(-3456)
%3 = [0, 1, -24, 252, -1472, 4830]
```

mfcoef(F, n)

Compute the *n*-th Fourier coefficient a(n) of the generalized modular form *F*. Note that this is the n + 1-st component of the vector mfcoefs(F,n) as well as the second component of mfcoefs(F,1,n).

? mfcoef(mfDelta(),10)
%1 = -115920

mfcoefs(F, n, d)

Compute the vector of Fourier coefficients [a[0], a[d], ..., a[nd]] of the generalized modular form F; d must be positive and d = 1 by default.

```
? D = mfDelta();
? mfcoefs(D,10)
%2 = [0, 1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
```

```
? mfcoefs(D,5,2)
%3 = [0, -24, -1472, -6048, 84480, -115920]
? mfcoef(D,10)
%4 = -115920
```

This function also applies when F is a modular form space as output by mfinit; it then returns the matrix whose columns give the Fourier expansions of the elements of mfbasis(F):

```
? mf = mfinit([1,12]);
? mfcoefs(mf,5)
%2 =
[691/65520 0]
[ 1 1]
[ 2049 -24]
[ 177148 252]
[ 4196353 -1472]
[ 48828126 4830]
```

mfconductor(mf, F)

mf being output by mfinit for the cuspidal space and F a modular form, gives the smallest level at which F is defined. In particular, if F is cuspidal and we write $F = \sum_j B(d_j)f_j$ for new forms f_j of level N_j (see mftonew), then its conductor is the least common multiple of the $d_j N_j$.

```
? mf=mfinit([96,6],1); vF = mfbasis(mf); mfdim(mf)
%1 = 72
? vector(10,i, mfconductor(mf, vF[i]))
%2 = [3, 6, 12, 24, 48, 96, 4, 8, 12, 16]
```

mfcosets(N)

Let N be a positive integer. Return the list of right cosets of $\Gamma_0(N)$ Gamma, i.e., matrices $\gamma_j \in \Gamma$ such that $\Gamma = \bigsqcup_i \Gamma_0(N)\gamma_j$. The γ_j are chosen in the form [a, b; c, d] with c || N.

? mfcosets(4)
%1 = [[0, -1; 1, 0], [1, 0; 1, 1], [0, -1; 1, 2], [0, -1; 1, 3],\
[1, 0; 2, 1], [1, 0; 4, 1]]

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

? M = mfinit([4, 12, 1], 0); mfcosets(M)
%2 = [[0, -1; 1, 0], [1, 0; 1, 1], [0, -1; 1, 2], [0, -1; 1, 3],\
[1, 0; 2, 1], [1, 0; 4, 1]]

Warning. In the present implementation, the trivial coset is represented by [1,0; N, 1] and is the last in the list.

mfcuspisregular(NK, cusp)

In the space defined by NK = [N, k, CHI] or NK = mf, determine if cusp in canonical format (oo or denominator dividing N) is regular or not.

? mfcuspisregular([4,3,-4],1/2)
%1 = 0

mfcusps(N)

Let N be a positive integer. Return the list of cusps of $\Gamma_0(N)$ in the form a/b with b||N.

? mfcusps(24) %1 = [0, 1/2, 1/3, 1/4, 1/6, 1/8, 1/12, 1/24]

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

? M = mfinit([4, 12, 1], 0); mfcusps(M) %2 = [0, 1/2, 1/4]

mfcuspval(mf, F, cusp, precision)

Valuation of modular form F in the space mf at cusp, which can be either *oo* or any rational number. The result is either a rational number or *oo* if F is zero. Let χ be the Nebentypus of the space mf; if $\mathbb{Q}(F)! = \mathbb{Q}(\chi)$, return the vector of valuations attached to the $[\mathbb{Q}(F): \mathbb{Q}(chi)]$ complex embeddings of F.

```
? T=mfTheta(); mf=mfinit([12,1/2]); mfcusps(12)
%1 = [0, 1/2, 1/3, 1/4, 1/6, 1/12]
? apply(x->mfcuspval(mf,T,x), %1)
\%2 = [0, 1/4, 0, 0, 1/4, 0]
? mf=mfinit([12,6,12],1); F=mfbasis(mf)[5];
? apply(x->mfcuspval(mf,F,x),%1)
\%4 = [1/12, 1/6, 1/2, 2/3, 1/2, 2]
? mf=mfinit([12,3,-4],1); F=mfbasis(mf)[1];
? apply(x->mfcuspval(mf,F,x),%1)
%6 = [1/12, 1/6, 1/4, 2/3, 1/2, 1]
? mf = mfinit([625,2],0); [F] = mfeigenbasis(mf); mfparams(F)
\%7 = [625, 2, 1, y^2 - y - 1, t - 1] \setminus [Q(F):Q(chi)] = 2
? mfcuspval(mf, F, 1/25)
\% = [1, 2] \\ one conjugate has valuation 1, and the other is 2
? mfcuspval(mf, F, 1/5)
\%9 = [1/25, 1/25]
```

mfcuspwidth(N, cusp)

Width of cusp in $\Gamma_0(N)$.

? mfcusps(12) %1 = [0, 1/2, 1/3, 1/4, 1/6, 1/12] ? [mfcuspwidth(12,c) | c <- mfcusps(12)] %2 = [12, 3, 4, 3, 1, 1] ? mfcuspwidth(12, oo) %3 = 1

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

? M = mfinit([4, 12, 1], 0); mfcuspwidth(M, 1/2) %4 = 1

mfderiv(F, m)

m-th formal derivative of the power series corresponding to the generalized modular form *F*, with respect to the differential operator qd/dq (default m = 1).

```
? D=mfDelta();
? mfcoefs(D, 4)
%2 = [0, 1, -24, 252, -1472]
? mfcoefs(mfderiv(D), 4)
%3 = [0, 1, -48, 756, -5888]
```

mfderivE2(F, m)

Compute the Serre derivative $(q.d/dq)F - kE_2F/12$ of the generalized modular form F, which has weight k+2; if F is a true modular form, then its Serre derivative is also modular. If m > 1, compute the m-th iterate, of weight k + 2m.

```
? mfcoefs(mfderivE2(mfEk(4)),5)*(-3)
%1 = [1, -504, -16632, -122976, -532728]
? mfcoefs(mfEk(6),5)
%2 = [1, -504, -16632, -122976, -532728]
```

mfdescribe(F, G)

Gives a human-readable description of F, which is either a modular form space or a generalized modular form. If the address of G is given, puts into G the vector of parameters of the outermost operator defining F; this vector is empty if F is a leaf (an atomic object such as mfDelta(), not defined in terms of other forms) or a modular form space.

```
? E1 = mfeisenstein(4,-3,-4); mfdescribe(E1)
\%1 = "F_4(-3, -4)"
? E2 = mfeisenstein(3,5,-7); mfdescribe(E2)
\%2 = "F_3(5, -7)"
? E3 = mfderivE2(mfmul(E1,E2), 3); mfdescribe(E3,&G)
\%3 = "DERE2^3(MUL(F_4(-3, -4), F_3(5, -7)))"
? mfdescribe(G[1][1])
%4 = "MUL(F_4(-3, -4), F_3(5, -7))"
? G[2]
%5 = 3
? for (i = 0, 4, mf = mfinit([37,4],i); print(mfdescribe(mf)));
S_4^{new}(G_0(37, 1))
S_4(G_0(37, 1))
S_4^old(G_0(37, 1))
E_4(G_0(37, 1))
M_4(G_0(37, 1))
```

mfdim(NK, space)

If NK = [N, k, CHI] as in mfinit, gives the dimension of the corresponding subspace of $M_k(\Gamma_0(N), \chi)$. NK can also be the output of mfinit, in which case space must be omitted.

The subspace is described by the small integer space: 0 for the newspace $S_k^{new}(\Gamma_0(N), \chi)$, 1 for the cuspidal space S_k , 2 for the oldspace S_k^{old} , 3 for the space of Eisenstein series E_k and 4 for the full space M_k .

Wildcards. As in mfinit, *CHI* may be the wildcard 0 (all Galois orbits of characters); in this case, the output is a vector of [*order*, *conrey*, *dim*, *dimdih*] corresponding to the nontrivial spaces, where

- order is the order of the character,
- *conrey* its Conrey label from which the character may be recovered via znchar(*conrey*),

- *dim* the dimension of the corresponding space,
- *dimdih* the dimension of the subspace of dihedral forms corresponding to Hecke characters if k = 1 (this is not implemented for the old space and set to -1 for the time being) and 0 otherwise.

The spaces are sorted by increasing order of the character; the characters are taken up to Galois conjugation and the Conrey number is the minimal one among Galois conjugates. In weight 1, this is only implemented when the space is 0 (newspace), 1 (cusp space), 2(old space) or 3(Eisenstein series).

Wildcards for sets of characters. CHI may be a set of characters, and we return the set of [dim, dimdih].

Wildcard for :math: $M_k(Gamma_1(N))$. Additionally, the wildcard CHI = -1 is available in which case we output the total dimension of the corresponding subspace of $M_k(\Gamma_1(N))$. In weight 1, this is not implemented when the space is 4 (fullspace).

```
? mfdim([23,2], 0) \\ new space
%1 = 2
? mfdim([96,6], 0)
%2 = 10
? mfdim([10^9,4], 3) \\ Eisenstein space
%1 = 40000
? mfdim([10^9+7,4], 3)
%2 = 2
? mfdim([68,1,-1],0)
%3 = 3
? mfdim([68,1,0],0)
%4 = [[2, Mod(67, 68), 1, 1], [4, Mod(47, 68), 1, 1]]
? mfdim([124,1,0],0)
%5 = [[6, Mod(67, 124), 2, 0]]
```

This last example shows that there exists a nondihedral form of weight 1 in level 124.

mfdiv(F, G)

Given two generalized modular forms F and G, compute F/G assuming that the quotient will not have poles at infinity. If this is the case, use mfshift before doing the division.

```
? D = mfDelta(); \\ Delta
? H = mfpow(mfEk(4), 3);
? J = mfdiv(H, D)
*** at top-level: J=mfdiv(H,mfdeltac
*** ^------
*** mfdiv: domain error in mfdiv: ord(G) > ord(F)
? J = mfdiv(H, mfshift(D,1));
? mfcoefs(J, 4)
%4 = [1, 744, 196884, 21493760, 864299970]
```

mfeigenbasis(mf)

Vector of the eigenforms for the space mf. The initial basis of forms computed by mfinit before splitting is also available via mfbasis.

```
? mf = mfinit([26,2],0);
? see(L) = for(i=1,#L,print(mfcoefs(L[i],6)));
? see( mfeigenbasis(mf) )
[0, 1, -1, 1, 1, -3, -1]
[0, 1, 1, -3, 1, -1, -3]
? see( mfbasis(mf) )
```

[0, 2, 0, -2, 2, -4, -4][0, -2, -4, 10, -2, 0, 8]

The eigenforms are internally expressed as (algebraic) linear combinations of mfbasis(mf) and it is very inefficient to compute many coefficients of those forms individually: you should rather use mfcoefs(mf) to expand the basis once and for all, then multiply by mftobasis(mf, f) for the forms you're interested in:

? mf = mfinit([96,6],0); B = mfeigenbasis(mf); #B %1 = 8; ? vector(#B, i, mfcoefs(B[i],1000)); \\ expanded individually: slow time = 7,881 ms. ? M = mfcoefs(mf, 1000); \\ initialize once time = 982 ms. ? vector(#B, i, M * mftobasis(mf,B[i])); \\ then expand: much faster time = 623 ms.

When the eigenforms are defined over an extension field of $\mathbb{Q}(\chi)$ for a nonrational character, their coefficients are hard to read and you may want to lift them or to express them in an absolute number field. In the construction below T defines $\mathbb{Q}(f)$ over \mathbb{Q} , a is the image of the generator $Mod(t, t^2 + t + 1)$ of $\mathbb{Q}(\chi)$ in $\mathbb{Q}(f)$ and y - ka is the image of the root y of f.mod:

```
? mf = mfinit([31, 2, Mod(25,31)], 0); [f] = mfeigenbasis(mf);
? f.mod
%2 = Mod(1, t<sup>2</sup> + t + 1)*y<sup>2</sup> + Mod(2*t + 2, t<sup>2</sup> + t + 1)
? v = liftpol(mfcoefs(f,5))
%3 = [0, 1, (-t - 1)*y - 1, t*y + (t + 1), (2*t + 2)*y + 1, t]
? [T,a,k] = rnfequation(mf.mod, f.mod, 1)
%4 = [y<sup>4</sup> + 2*y<sup>2</sup> + 4, Mod(-1/2*y<sup>2</sup> - 1, y<sup>4</sup> + 2*y<sup>2</sup> + 4), 0]
? liftpol(substvec(v, [t,y], [a, y-k*a]))
%5 = [0, 1, 1/2*y<sup>3</sup> - 1, -1/2*y<sup>3</sup> - 1/2*y<sup>2</sup> - y, -y<sup>3</sup> + 1, -1/2*y<sup>2</sup> - 1]
```

Beware that the meaning of y has changed in the last line is different: it now represents of root of T, no longer of f.mod (the notions coincide if k = 0 as here but it will not always be the case). This can be avoided with an extra variable substitution, for instance

```
? [T,a,k] = rnfequation(mf.mod, subst(f.mod,'y,'x), 1)
%6 = [x^4 + 2*x^2 + 4, Mod(-1/2*x^2 - 1, x^4 + 2*x^2 + 4), 0]
? liftpol(substvec(v, [t,y], [a, x-k*a]))
%7 = [0, 1, 1/2*x^3 - 1, -1/2*x^3 - 1/2*x^2 - x, -x^3 + 1, -1/2*x^2 - 1]
```

mfeigensearch(NK, AP)

Search for a normalized rational eigen cuspform with quadratic character given restrictions on a few initial coefficients. The meaning of the parameters is as follows:

- NK governs the limits of the search: it is of the form [N, k]: search for given level N, weight k and quadratic character; note that the character (D/.) is uniquely determined by (N, k). The level N can be replaced by a vector of allowed levels.
- AP is the search criterion, which can be omitted: a list of pairs $[..., [p, a_p], ...]$, where p is a prime number and a_p is either a t_INT (the p-th Fourier coefficient must match a_p exactly) or a t_INTMOD Mod(a, b) (the p-th coefficient must be congruent to a modulo b).

The result is a vector of newforms f matching the search criteria, sorted by increasing level then increasing ||D||.

? #mfeigensearch([[1..80],2], [[2,2],[3,-1]]) %1 = 1 ? #mfeigensearch([[1..80],2], [[2,2],[5,2]]) %2 = 1 ? v = mfeigensearch([[1..20],2], [[3,Mod(2,3)],[7,Mod(5,7)]]); #v %3 = 1 ? F=v[1]; [mfparams(F)[1], mfcoefs(F,15)] %4 = [11, [0, 1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1]]

mfeisenstein(k, CHI1, CHI2)

Create the Eisenstein series $E_k(\chi_1, \chi_2)$, where $k \ge 1$, χ_i are Dirichlet characters and an omitted character is considered as trivial. This form belongs to $E_k(\Gamma_0(N), \chi)$ with $\chi = \chi_1 \chi_2$ and N is the product of the conductors of χ_1 and χ_2 .

```
? CHI = Mod(3,4);
? E = mfeisenstein(3, CHI);
? mfcoefs(E, 6)
%2 = [-1/4, 1, 1, -8, 1, 26, -8]
? CHI2 = Mod(4,5);
? mfcoefs(mfeisenstein(3,CHI,CHI2), 6)
%3 = [0, 1, -1, -10, 1, 25, 10]
? mfcoefs(mfeisenstein(4,CHI,CHI), 6)
%4 = [0, 1, 0, -28, 0, 126, 0]
? mfcoefs(mfeisenstein(4), 6)
%5 = [1/240, 1, 9, 28, 73, 126, 252]
```

Note that mfeisenstein(k) is 0 for k odd and $-B_k/(2k) \cdot E_k$ for k even, where

$$E_k(q) = 1 - (2k/B_k) \sum_{n>=1} \sigma_{k-1}(n)q^n$$

is the standard Eisenstein series. In other words it is normalized so that its linear coefficient is 1.

Important note. This function is currently implemented only when $\mathbb{Q}(\chi)$ is the field of definition of $E_k(\chi_1, \chi_2)$. If it is a strict subfield, an error is raised:

```
? mfeisenstein(6, Mod(7,9), Mod(4,9));
  *** at top-level: mfeisenstein(6,Mod(7,9),Mod(4,9))
  *** ^-------
  *** mfeisenstein: sorry, mfeisenstein for these characters is not
  *** yet implemented.
```

The reason for this is that each modular form is attached to a modular form space $M_k(\Gamma_0(N), \chi)$. This is a \mathbb{C} -vector space but it allows a basis of forms defined over $\mathbb{Q}(\chi)$ and is only implemented as a $\mathbb{Q}(\chi)$ -vector space: there is in general no mechanism to take linear combinations of forms in the space with coefficients belonging to a larger field. (Due to their importance, eigenforms are the single exception to this restriction; for an eigenform $F, \mathbb{Q}(F)$ is built on top of $\mathbb{Q}(\chi)$.) When the property $\mathbb{Q}(\chi) = \mathbb{Q}(E_k(\chi_1, \chi_2))$ does not hold, we cannot express E as a $\mathbb{Q}(\chi)$ -linear combination of the basis forms and many operations will fail. For this reason, the construction is currently disabled.

mfembed(f, v, precision)

Let f be a generalized modular form with parameters $[N, k, \chi, P]$ (see mfparams, we denote $\mathbb{Q}(\chi)$ the subfield of \mathbb{C} generated by the values of χ and $\mathbb{Q}(f)$ the field of definition of f. In this context $\mathbb{Q}(\chi)$ has a single canonical complex embeding given by $s : Mod(t, polcyclo(n, t)) : -- - > \exp(2i\pi/n)$ and the number field $\mathbb{Q}(f)$ has $[\mathbb{Q}(f) : \mathbb{Q}(\chi)]$ induced embeddings attached to the complex roots of the polynomial s(P). If $\mathbb{Q}(f)$ is strictly larger than $\mathbb{Q}(\chi)$ we only allow an f which is an eigenform, produced by mfeigenbasis.

This function is meant to create embeddings of $\mathbb{Q}(f)$ and/or apply them to the object v, typically a vector of Fourier coefficients of f from mfcoefs.

- If v is omitted and f is a modular form as above, we return the embedding of Q(χ) if Q(χ) = Q(f) and a vector containing [Q(f) : Q(χ)] embeddings of Q(f) otherwise.
- If v is given, it must be a scalar in Q(f), or a vector/matrix of such, we apply the embeddings coefficientwise and return either a single result if Q(f) = Q(χ) and a vector of [Q(f) : Q(χ)] results otherwise.
- Finally f can be replaced by a single embedding produced by mfembed(f) (v was omitted) and we apply that particular embedding to v.

```
? mf = mfinit([35,2,Mod(11,35)], 0);
? [f] = mfbasis(mf);
? f.mod \setminus Q (chi) = Q (zeta_3)
\%3 = t^2 + t + 1
? v = mfcoefs(f,5); lift(v) \\ coefficients in Q (chi)
%4 = [0, 2, -2*t - 2, 2*t, 2*t, -2*t - 2]
? mfembed(f, v) \setminus single embedding
%5 = [0, 2, -1 - 1.7320...*I, -1 + 1.73205...*I, -1 + 1.7320...*I, ...]
? [F] = mfeigenbasis(mf);
? mffields(mf)
\%7 = [y^2 + Mod(-2*t, t^2 + t + 1)] \setminus [Q (f):Q (chi)] = 2
? V = liftpol( mfcoefs(F,5) );
\%8 = [0, 1, y + (-t - 1), (t + 1)*y + t, (-2*t - 2)*y + t, -t - 1]
? vall = mfembed(F, V); #vall
\%9 = 2 \setminus 2 embeddings, both applied to V
? vall[1] \setminus the first
%10 = [0, 1, -1.2071... - 2.0907...*I, 0.2071... - 0.3587...*I, ...]
? vall[2] \setminus and the second one
%11 = [0, 1, 0.2071... + 0.3587...*I, -1.2071... + 2.0907...*I, ...]
? vE = mfembed(F); #vE \\ same 2 embeddings
\%12 = 2
? mfembed(vE[1], V) \\ apply first embedding to V
%13 = [0, 1, -1.2071... - 2.0907...*I, 0.2071... - 0.3587...*I, ...]
```

For convenience, we also allow a modular form space from mfinit instead of f, corresponding to the single embedding of $\mathbb{Q}(\chi)$.

```
? [mfB,MC,C] = mfatkininit(mf,7); MC \\ coefs in Q (chi)
%13 =
[ Mod(2/7*t, t^2 + t + 1) Mod(-1/7*t - 2/7, t^2 + t + 1)]
[Mod(-1/7*t - 2/7, t^2 + t + 1) Mod(2/7*t, t^2 + t + 1)]
? C \\ normalizing constant
%14 = 0.33863... - 0.16787*I
? M = mfembed(mf, MC) / C \\ the true matrix for the action of w_7
[-0.6294... + 0.4186...*I -0.3625... - 0.5450...*I]
[-0.3625... - 0.5450...*I -0.6294... + 0.4186...*I]
```

```
? exponent(M*conj(M) - 1) \\ M * conj(M) is close to 1
%16 = -126
```

mfeval(*mf*, *F*, *vtau*, *precision*)

Computes the numerical value of the modular form F, belonging to mf, at the complex number vtau or the vector vtau of complex numbers in the completed upper-half plane. The result is given with absolute error less than 2^{-B} , where B = realbit precision.

If the field of definition $\mathbb{Q}(F)$ is larger than $\mathbb{Q}(\chi)$ then F may be embedded into \mathbb{C} in $d = [\mathbb{Q}(F) : \mathbb{Q}(\chi)]$ ways, in which case a vector of the d results is returned.

```
? mf = mfinit([11,2],0); F = mfbasis(mf)[1]; mfparams(F)
%1 = [11, 2, 1, y, t-1] \\ Q(F) = Q(chi) = Q
? mfeval(mf,F,I/2)
%2 = 0.039405471130100890402470386372028382117
? mf = mfinit([35,2],0); F = mfeigenbasis(mf)[2]; mfparams(F)
%3 = [35, 2, 1, y^2 - y - 4, t - 1] \\ [Q(F) : Q(chi)] = 2
? mfeval(mf,F,I/2)
%4 = [0.045..., 0.0385...] \\ sigma_1(F) and sigma_2(F) at I/2
? mf = mfinit([12,4],1); F = mfbasis(mf)[1];
? mfeval(mf, F, 0.318+10^(-7)*I)
%6 = 3.379... E-21 + 6.531... E-21*I \\ instantaneous !
```

In order to maximize the imaginary part of the argument, the function computes $(f||_k\gamma)(\gamma^{-1}.\tau)$ for a suitable γ not necessarily in $\Gamma_0(N)$ (in which case $f||\gamma$ is evaluated using mfslashexpansion).

```
? T = mfTheta(); mf = mfinit(T); mfeval(mf,T,[0,1/2,1,oo])
%1 = [1/2 - 1/2*I, 0, 1/2 - 1/2*I, 1]
```

mffields(mf)

Given mf as output by mfinit with parameters (N, k, χ) , returns the vector of polynomials defining each Galois orbit of newforms over $\mathbb{Q}(\chi)$.

```
? mf = mfinit([35,2],0); mffields(mf)
%1 = [y, y^2 - y - 4]
```

Here the character is trivial so $\mathbb{Q}(\chi) = \mathbb{Q}$) and there are 3 newforms: one is rational (corresponding to y), the other two are conjugate and defined over the quadratic field $\mathbb{Q}[y]/(y^2 - y - 4)$.

```
? [G,chi] = znchar(Mod(3,35));
? zncharconductor(G,chi)
%2 = 35
? charorder(G,chi)
%3 = 12
? mf = mfinit([35, 2, [G,chi]],0); mffields(mf)
%4 = [y, y]
```

Here the character is primitive of order 12 and the two newforms are defined over $\mathbb{Q}(\chi) = \mathbb{Q}(\zeta_{12})$.

```
? mf = mfinit([35, 2, Mod(13,35)],0); mffields(mf)
%3 = [y^2 + Mod(5*t, t^2 + 1)]
```

This time the character has order 4 and there are two conjugate newforms over $\mathbb{Q}(\chi) = Q(i)$.

mffromell(E)

E being an elliptic curve defined over Q given by an integral model in ellipsit format, computes a 3-component vector [mf,F,v], where *F* is the newform corresponding to *E* by modularity, mf is the newspace to which *F* belongs, and v gives the coefficients of *F* on mfbasis(mf).

```
? E = ellinit("26a1");
? [mf,F,co] = mffromell(E);
? co
%2 = [3/4, 1/4]~
? mfcoefs(F, 5)
%3 = [0, 1, -1, 1, 1, -3]
? ellan(E, 5)
%4 = [1, -1, 1, 1, -3]
```

mffrometaquo(eta, flag)

Modular form corresponding to the eta quotient matrix eta. If the valuation v at infinity is fractional, return 0. If the eta quotient is not holomorphic but simply meromorphic, return 0 if flag = 0; return the eta quotient (divided by q to the power -v if v < 0, i.e., with valuation 0) if flag is set.

```
? mffrometaquo(Mat([1,1]),1)
%1 = 0
? mfcoefs(mffrometaquo(Mat([1,24])),6)
%2 = [0, 1, -24, 252, -1472, 4830, -6048]
? mfcoefs(mffrometaquo([1,1;23,1]),10)
%3 = [0, 1, -1, -1, 0, 0, 1, 0, 1, 0, 0]
? F = mffrometaquo([1,2;2,-1]); mfparams(F)
%4 = [16, 1/2, 1, y, t - 1]
? mfcoefs(F,10)
%5 = [1, -2, 0, 0, 2, 0, 0, 0, 0, -2, 0]
? mffrometaquo(Mat([1,-24]))
%6 = 0
? f = mffrometaquo(Mat([1,-24]),1); mfcoefs(f,6)
%7 = [1, 24, 324, 3200, 25650, 176256, 1073720]
```

For convenience, a t_VEC is also accepted instead of a factorization matrix with a single row:

```
? f = mffrometaquo([1,24]); \\ also valid
```

mffromlfun(*L*, *precision*)

Let *L* being an *L*-function in any of the lfun formats representing a self-dual modular form (for instance an eigenform). Return [NK, space, v] when mf = mfinit(NK, space) is the modular form space containing the form and mftobasis(mf, v) will represent it on the space basis. If *L* has rational coefficients, this will be enough to recognize the modular form in *mf*:

If L has inexact complex coefficients, one can for instance compute an eigenbasis for mf and check whether one of the attached L-function is reasonably close to L. In the example, we cheat by producing the L function from an eigenform in a known space, but the function does not use this information:

? mf = mfinit([32,6,Mod(5,32)],0); ? [poldegree(K) | K<-mffields(mf)] %2 = [19] \\ one orbit, [Q(F) : Q(chi)] = 19 ? L = lfunmf(mf)[1][1]; \\ one of the 19 L-functions attached to F ? lfunan(L,3) %4 = [1, 5.654... - 0.1812...*I, -7.876... - 19.02...*I] ? [NK,space,v] = mffromlfun(L); NK %5 = [32, 6, Mod(5, 32)] ? vL = concat(lfunmf(mf)); \\ L functions for all cuspidal eigenforms ? an = lfunan(L,10); ? for (i = 1, #vL, if (normlp(lfunan(vL[i],10) - an, oo) < 1e-10, print(i))); 1

mffromqf(Q, P)

Q being an even integral positive definite quadratic form and P a homogeneous spherical polynomial for Q, computes a 3-component vector [mf, F, v], where F is the theta function corresponding to (Q, P), mf is the corresponding space of modular forms (from mfinit), and v gives the coefficients of F on mfbasis(mf).

```
? [mf,F,v] = mffromqf(2*matid(10)); v
%1 = [64/5, 4/5, 32/5]~
? mfcoefs(F, 5)
%2 = [1, 20, 180, 960, 3380, 8424]
? mfcoef(F, 10000) \\ number of ways of writing 10000 as sum of 10 squares
%3 = 128205250571893636
? mfcoefs(F, 10000); \\ fast !
time = 220ms
? [mf,F,v] = mffromqf([2,0;0,2],x^4-6*x^2*y^2+y^4);
? mfcoefs(F,10)
%6 = [0, 4, -16, 0, 64, -56, 0, 0, -256, 324, 224]
? mfcoef(F,100000) \\ instantaneous
%7 = 41304367104
```

Odd dimensions are supported, corresponding to forms of half-integral weight:

```
? [mf,F,v] = mffromqf(2*matid(3));
? mfisequal(F, mfpow(mfTheta(),3))
%2 = 1
? mfcoefs(F, 32) \\ illustrate Legendre's 3-square theorem
%3 = [ 1,
6, 12, 8, 6, 24, 24, 0, 12,
30, 24, 24, 8, 24, 48, 0, 6,
48, 36, 24,24, 48, 24, 0, 24,
30, 72, 32, 0, 72, 48, 0, 12]
```

mfgaloisprojrep(mf, F, precision)

mf being an mf output by mfinit in weight 1, return a polynomial defining the field fixed by the kernel of the projective Artin representation attached to F (by Deligne-Serre). Currently only implemented for projective image A_4 and S_4 .

```
\\ A4 example
? mf = mfinit([4*31,1,Mod(87,124)],0);
? F = mfeigenbasis(mf)[1];
? mfgaloistype(mf,F)
\%3 = -12
? pol = mfgaloisprojrep(mf,F)
\%4 = x^{12} + 68 * x^{10} + 4808 * x^{8} + \ldots + 4096
? G = galoisinit(pol); galoisidentify(G)
\%5 = [12,3] \setminus A4
? pol4 = polredbest(galoisfixedfield(G,G.gen[3], 1))
\%6 = x^4 + 7^*x^2 - 2^*x + 14
? polgalois(pol4)
\%7 = [12, 1, 1, "A4"]
? factor(nfdisc(pol4))
%8 =
[24]
[31 2]
\setminus S4 example
? mf = mfinit([4*37,1,Mod(105,148)],0);
? F = mfeigenbasis(mf)[1];
? mfgaloistype(mf,F)
\%11 = -24
? pol = mfgaloisprojrep(mf,F)
12 = x^24 + 24^*x^22 + 256^*x^20 + \ldots + 255488256
? G = galoisinit(pol); galoisidentify(G)
%13 = [24, 12] \setminus S4
? pol4 = polredbest(galoisfixedfield(G,G.gen[3..4], 1))
\%14 = x^{4} - x^{3} + 5*x^{2} - 7*x + 12
? polgalois(pol4)
%15 = [24, -1, 1, "S4"]
? factor(nfdisc(pol4))
%16 =
[22]
[37 3]
```

mfgaloistype(NK, F)

NK being either [N, 1, CHI] or an mf output by mfinit in weight 1, gives the vector of types of Galois representations attached to each cuspidal eigenform, unless the modular form F is specified, in which case only for F (note that it is not tested whether F belongs to the correct modular form space, nor whether it is a cuspidal eigenform). Types A_4 , S_4 , A_5 are represented by minus their cardinality -12, -24, or -60, and type D_n is represented by its cardinality, the integer 2n:

```
? mfgaloistype([124,1, Mod(67,124)]) \\ A4
%1 = [-12]
? mfgaloistype([148,1, Mod(105,148)]) \\ S4
%2 = [-24]
? mfgaloistype([633,1, Mod(71,633)]) \\ D10, A5
%3 = [10, -60]
? mfgaloistype([239,1, -239]) \\ D6, D10, D30
%4 = [6, 10, 30]
```

```
? mfgaloistype([71,1, -71])
%5 = [14]
? mf = mfinit([239,1, -239],0); F = mfeigenbasis(mf)[2];
? mfgaloistype(mf, F)
%7 = 10
```

The function may also return 0 as a type when it failed to determine it; in this case the correct type is either -12 or -60, and most likely -12.

mfnecke(mf, F, n)

F being a modular form in modular form space mf, returns T(n)F, where T(n) is the n-th Hecke operator.

Warning. If F is of level M < N, then T(n)F is in general not the same in $M_k(\Gamma_0(M), \chi)$ and in $M_k(\Gamma_0(N), \chi)$. We take T(n) at the same level as the one used in mf.

```
? mf = mfinit([26,2],0); F = mfbasis(mf)[1]; mftobasis(mf,F)
%1 = [1, 0]~
? G2 = mfhecke(mf,F,2); mftobasis(mf,G2)
%2 = [0, 1]~
? G5 = mfhecke(mf,F,5); mftobasis(mf,G5)
%3 = [-2, 1]~
```

Modular forms of half-integral weight are supported, in which case n must be a perfect square, else T_n will act as 0 (the operator T_p for p || N is not supported yet):

? F = mfpow(mfTheta(),3); mf = mfinit(F); ? mfisequal(mfhecke(mf,F,9), mflinear([F],[4])) %2 = 1

(F is an eigenvector of all T_{p^2} , with eigenvalue p + 1 for odd p.)

Warning. When n is a large composite, resp. the square of a large composite in half-integral weight, it is in general more efficient to use mfheckemat on the mftobasis coefficients:

```
? mfcoefs(mfhecke(mf,F,3^10), 10)
time = 917 ms.
%3 = [324, 1944, 3888, 2592, 1944, 7776, 7776, 0, 3888, 9720, 7776]
? M = mfheckemat(mf,3^10) \\ instantaneous
%4 =
[324]
? G = mflinear(mf, M*mftobasis(mf,F));
? mfcoefs(G, 10) \\ instantaneous
%6 = [324, 1944, 3888, 2592, 1944, 7776, 7776, 0, 3888, 9720, 7776]
```

mfheckemat(mf, vecn)

If vecn is an integer, matrix of the Hecke operator T(n) on the basis formed by mfbasis(mf). If it is a vector, vector of such matrices, usually faster than calling each one individually.

? mf=mfinit([32,4],0); mfheckemat(mf,3)
%1 =
[0 44 0]
[1 0 -10]

[0 -2 0] ? mfheckemat(mf,[5,7]) %2 = [[0, 0, 220; 0, -10, 0; 1, 0, 12], [0, 88, 0; 2, 0, -20; 0, -4, 0]]

mfinit(NK, space)

Create the space of modular forms corresponding to the data contained in NK and space. NK is a vector which can be either [N, k] (N level, k weight) corresponding to a subspace of $M_k(\Gamma_0(N))$, or [N, k, CHI] (CHI a character) corresponding to a subspace of $M_k(\Gamma_0(N), \chi)$. Alternatively, it can be a modular form F or modular form space, in which case we use mfparams to define the space parameters.

The subspace is described by the small integer space: 0 for the newspace $S_k^{new}(\Gamma_0(N), \chi)$, 1 for the cuspidal space S_k , 2 for the oldspace S_k^{old} , 3 for the space of Eisenstein series E_k and 4 for the full space M_k .

Wildcards. For given level and weight, it is advantageous to compute simultaneously spaces attached to different Galois orbits of characters, especially in weight 1. The parameter *CHI* may be set to 0 (wildcard), in which case we return a vector of all mfinit (s) of non trivial spaces in $S_k(\Gamma_1(N))$, one for each Galois orbit (see znchargalois). One may also set *CHI* to a vector of characters and we return a vector of all mfinits of subspaces of $M_k(G_0(N), \chi)$ for χ in the list, in the same order. In weight 1, only S_1^{new} , S_1 and E_1 support wildcards.

The output is a technical structure S, or a vector of structures if *CHI* was a wildcard, which contains the following information: $[N, k, \chi]$ is given by mfparams(S), the space dimension is mfdim(S) and a \mathbb{C} -basis for the space is mfbasis(S). The structure is entirely algebraic and does not depend on the current realbitprecision.

```
? S = mfinit([36,2], 0); \\ new space
? mfdim(S)
%2 = 1
? mfparams
%3 = [36, 2, 1, y] \\ trivial character
? f = mfbasis(S)[1]; mfcoefs(f,10)
%4 = [0, 1, 0, 0, 0, 0, 0, -4, 0, 0, 0]
? vS = mfinit([36,2,0],0); \\ with wildcard
? #vS
%6 = 4 \\ 4 non trivial spaces (mod Galois action)
? apply(mfdim,vS)
%7 = [1, 2, 1, 4]
? mfdim([36,2,0], 0)
%8 = [[1, Mod(1, 36), 1, 0], [2, Mod(35, 36), 2, 0], [3, Mod(13, 36), 1, 0],
[6, Mod(11, 36), 4, 0]]
```

mfisCM(F)

Tests whether the eigenform F is a CM form. The answer is 0 if it is not, and if it is, either the unique negative discriminant of the CM field, or the pair of two negative discriminants of CM fields, this latter case occurring only in weight 1 when the projective image is $D_2 = C_2 x C_2$, i.e., coded 4 by mfgaloistype.

```
? F = mffromell(ellinit([0,1]))[2]; mfisCM(F)
%1 = -3
? mf = mfinit([39,1,-39],0); F=mfeigenbasis(mf)[1]; mfisCM(F)
%2 = Vecsmall([-3, -39])
? mfgaloistype(mf)
%3 = [4]
```

mfisequal(F, G, lim)

Checks whether the modular forms F and G are equal. If $\lim is$ nonzero, only check equality of the first lim + 1

Fourier coefficients and the function then also applies to generalized modular forms.

```
? D = mfDelta(); F = mfderiv(D);
? G = mfmul(mfEk(2), D);
? mfisequal(F, G)
%2 = 1
```

mfisetaquo(f, flag)

If the generalized modular form f is a holomorphic eta quotient, return the eta quotient matrix, else return 0. If *flag* is set, also accept meromorphic eta quotients: check whether $f = q^{-\nu(g)}g(q)$ for some eta quotient g; if so, return the eta quotient matrix attached to g, else return 0. See mffrometaquo.

```
? mfisetaquo(mfDelta())
%1 =
[1 24]
? f = mffrometaquo([1,1;23,1]);
? mfisetaquo(f)
%3 =
[ 1 1]
[23 1]
? f = mffrometaquo([1,-24], 1);
? mfisetaquo(f) \\ nonholomorphic
%5 = 0
? mfisetaquo(f,1)
%6 =
[1 -24]
```

mfkohnenbasis(mf)

mf being a cuspidal space of half-integral weight $k \ge 3/2$ with level N and character χ , gives a basis B of the Kohnen +-space of mf as a matrix whose columns are the coefficients of B on the basis of mf. The conductor of either χ or χ .(-4/.) must divide N/4.

In the final example both $\chi = (8/.)$ and $\chi \cdot (-4/.)$ have conductor 8, which does not divide N/4 = 10.

mfkohnenbijection(mf)

mf being a cuspidal space of half-integral weight, returns [mf2,M,K,shi], where M is a matrix giving a Hecke-module isomorphism from the cuspidal space mf2 giving $S_{2k-1}(\Gamma_0(N), \chi^2)$ to the Kohnen +-space

 $S_k^+(\Gamma_0(4N), \chi)$, K represents a basis B of the Kohnen +-space as a matrix whose columns are the coefficients of B on the basis of mf; shi is a vector of pairs (t_i, n_i) gives the linear combination of Shimura lifts giving M^{-1} : t_i is a squarefree positive integer and n_i is a small nonzero integer.

```
? mf=mfinit([60,5/2],1); [mf2,M,K,shi]=mfkohnenbijection(mf); M
%2 =
[-3 0 5/2 7/2]
[ 1 -1/2 -7 -7]
[ 1 1/2 0 -3]
[ 0 0 5/2 5/2]
? shi
%2 = [[1, 1], [2, 1]]
```

This last command shows that the map giving the bijection is the sum of the Shimura lift with t = 1 and the one with t = 2.

Since it gives a bijection of Hecke modules, this matrix can be used to transport modular form data from the easily computed space of level N and weight 2k - 1 to the more difficult space of level 4N and weight k: matrices of Hecke operators, new space, splitting into eigenspaces and eigenforms. Examples:

```
? K^(-1)*mfheckemat(mf,121)*K /* matrix of T_11^2 on K. Slowish. */
time = 1,280 ms.
%1 =
[ 48 24 24 24]
[ 0 32 0 -20]
[-48 -72 -40 -72]
[ 0 0 0 52]
? M*mfheckemat(mf2,11)*M^(-1) /* instantaneous via T_11 on S_{2k-1} */
time = 0 \text{ ms}.
%2 =
[ 48 24 24 24]
[ 0 32 0 -20]
[-48 -72 -40 -72]
[ 0 0 0 52]
? mf20=mfinit(mf2,0); [mftobasis(mf2,b) | b<-mfbasis(mf20)]</pre>
%3 = [[0, 0, 1, 0] \sim, [0, 0, 0, 1] \sim]
? F1=M*[0,0,1,0]~
\%4 = [1/2, 1/2, -3/2, -1/2] \sim
? F2=M*[0,0,0,1]~
\%5 = [3/2, 1/2, -9/2, -1/2]
? K*F1
%6 = [1, 0, 0, 1, 1, 0, 0, 1, -3, 0, 0, -3, 0, 0]~
? K*F2
%7 = [3, 0, 0, 3, 1, 0, 0, 1, -9, 0, 0, -3, 0, 0]~
```

This gives a basis of the new space of $S_{5/2}^+(\Gamma_0(60))$ expressed on the initial basis of $S_{5/2}(\Gamma_0(60))$. If we want the eigenforms, we write instead:

```
? BE=mfeigenbasis(mf20);[E1,E2]=apply(x->K*M*mftobasis(mf2,x),BE)
%1 = [[1, 0, 0, 1, 0, 0, 0, 0, -3, 0, 0, 0, 0, 0]~,\
[0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, -3, 0, 0]~,
? EI1 = mflinear(mf, E1); EI2=mflinear(mf, E2);
```

These are the two eigenfunctions in the space mf, the first (resp., second) will have Shimura image a multiple of BE[1] (resp., BE[2]). The function mfkohneneigenbasis does this directly.

mfkohneneigenbasis(mf, bij)

mf being a cuspidal space of half-integral weight $k \ge 3/2$ and bij being the output of mfkohnenbijection(mf), outputs a 3-component vector [mf0,BNEW,BEIGEN], where BNEW and BEIGEN are two matrices whose columns are the coefficients of a basis of the Kohnen new space and of the eigenforms on the basis of mf respectively, and mf0 is the corresponding new space of integral weight 2k - 1.

```
? mf=mfinit([44,5/2],1);bij=mfkohnenbijection(mf);
? [mf0,BN,BE]=mfkohneneigenbasis(mf,bij);
? BN~
%2 =
[2 0 0 -2 2 0 -8]
[2 0 0 4 14 0 -32]
? BE~
%3 = [1 0 0 Mod(y-1, y^2-3) Mod(2*y+1, y^2-3) 0 Mod(-4*y-4, y^2-3)]
? lift(mfcoefs(mf,20)*BE[,1])
%4 = [0, 1, 0, 0, y - 1, 2*y + 1, 0, 0, 0, -4*y - 4, 0, 0, \
-5*y + 3, 0, 0, 0, -6, 0, 0, 0, 7*y + 9]~
```

mflinear(vF, v)

vF being a vector of generalized modular forms and v a vector of coefficients of same length, compute the linear combination of the entries of vF with coefficients v. Note. Use this in particular to subtract two forms F and G (with vF = [F, G] and v = [1, -1]), or to multiply an form by a scalar λ (with vF = [F] and $v = [\lambda]$).

```
? D = mfDelta(); G = mflinear([D],[-3]);
? mfcoefs(G,4)
%2 = [0, -3, 72, -756, 4416]
```

For user convenience, we allow

- a modular form space mf as a vF argument, which is understood as mfbasis(mf);
- in this case, we also allow a modular form f as v, which is understood as mftobasis(mf, f).

```
? T = mfpow(mfTheta(),7); F = mfShimura(T,-3); \\ Shimura lift for D=-3
? mfcoefs(F,8)
%2 = [-5/9, 280, 9240, 68320, 295960, 875280, 2254560, 4706240, 9471000]
? mf = mfinit(F); G = mflinear(mf,F);
? mfcoefs(G,8)
%4 = [-5/9, 280, 9240, 68320, 295960, 875280, 2254560, 4706240, 9471000]
```

This last construction allows to replace a general modular form by a simpler linear combination of basis functions, which is often more efficient:

? T10=mfpow(mfTheta(),10); mfcoef(T10, 10^4) \\ direct evaluation time = 399 ms. %5 = 128205250571893636 ? mf=mfinit(T10); F=mflinear(mf,T10); \\ instantaneous ? mfcoef(F, 10^4) \\ after linearization time = 67 ms. %7 = 128205250571893636

mfmanin(FS, precision)

Given the modular symbol FS associated to an eigenform F by mfsymbol (mf,F), computes the even and odd special polynomials as well as the even and odd periods ω^+ and ω^- as a vector $[[P^+, P^-], [\omega^+, \omega^-, r]]$, where $r = \Im(\omega^+\overline{\omega^-})/\langle F, F \rangle$. If F has several embeddings into \mathbb{C} , give the vector of results corresponding to each embedding.

```
? D=mfDelta(); mf=mfinit(D); DS=mfsymbol(mf,D);
? [pols,oms]=mfmanin(DS); pols
%2 = [[4*x^9 - 25*x^7 + 42*x^5 - 25*x^3 + 4*x],\
    [-36*x^10 + 691*x^8 - 2073*x^6 + 2073*x^4 - 691*x^2 + 36]]
? oms
%3 = [0.018538552324740326472516069364750571812,\
    -0.00033105361053212432521308691198949874026*I, 4096/691]
? mf=mfinit([11,2],0); F=mfeigenbasis(mf)[1]; FS=mfsymbol(mf,F);
? [pols,oms]=mfmanin(FS);pols
%5 = [[0, 0, 0, 1, 1, 0, 0, -1, -1, 0, 0, 0],\
    [2, 0, 10, 5, -5, -10, -10, -5, 5, 10, 0, -2]]
? oms[3]
%6 = 24/5
```

mfmul(F, G)

Multiply the two generalized modular forms F and G.

```
? E4 = mfEk(4); G = mfmul(mfmul(E4,E4),E4);
? mfcoefs(G, 4)
%2 = [1, 720, 179280, 16954560, 396974160]
? mfcoefs(mfpow(E4,3), 4)
%3 = [1, 720, 179280, 16954560, 396974160]
```

mfnumcusps(N)

Number of cusps of $\Gamma_0(N)$

```
? mfnumcusps(24)
%1 = 8
? mfcusps(24)
%1 = [0, 1/2, 1/3, 1/4, 1/6, 1/8, 1/12, 1/24]
```

mfparams(F)

If F is a modular form space, returns [N,k,CHI,space,:math:Phi`], level, weight, character χ , and space code; where Φ is the cyclotomic polynomial defining the field of values of CHI. If F is a generalized modular form, returns [N,k,CHI,P,:math:Phi`], where P is the (polynomial giving the) field of definition of F as a relative extension of the cyclotomic field $\mathbb{Q}(\chi) = \mathbb{Q}[t]/(\Phi)$: in that case the level N may be a multiple of the level of F and the polynomial P may define a larger field than $\mathbb{Q}(F)$. If you want the true level of F from this result, use mfconductor(mfinit(F),F). The polynomial P defines an extension of $\mathbb{Q}(\chi) = \mathbb{Q}[t]/(\Phi(t))$; it has coefficients in that number field (polmods in t). In contrast with mfparams(F)[4] which always gives the polynomial P defining the relative extension $\mathbb{Q}(F)/\mathbb{Q}(\chi)$, the member function :math: F.mod` returns the polynomial used to define $\mathbb{Q}(F)$ over \mathbb{Q} (either a cyclotomic polynomial or a polynomial with cyclotomic coefficients).

```
? E1 = mfeisenstein(4,-3,-4); E2 = mfeisenstein(3,5,-7); E3 = mfmul(E1,E2);
? apply(mfparams, [E1,E2,E3])
%2 = [[12, 4, 12, y, t-1], [35, 3, -35, y, t-1], [420, 7, -420, y, t-1]]
? mf = mfinit([36,2,Mod(13,36)],0); [f] = mfeigenbasis(mf); mfparams(mf)
%3 = [36, 2, Mod(13, 36), 0, t^2 + t + 1]
? mfparams(f)
%4 = [36, 2, Mod(13, 36), y, t^2 + t + 1]
? f.mod
%5 = t^2 + t + 1
? mf = mfinit([36,4,Mod(13,36)],0); [f] = mfeigenbasis(mf);
? lift(mfparams(f))
%7 = [36, 4, 13, y^3 + (2*t-2)*y^2 + (-4*t+6)*y + (10*t-1), t^2+t+1]
```

mfperiodpol(mf, f, flag, precision)

Period polynomial of the cuspidal part of the form f, in other words $\int_0^{ioo} (X - \tau)^{k-2} f(\tau) d\tau$. If flag is 0, ordinary period polynomial. If it is 1 or -1, even or odd part of that polynomial. f can also be the modular symbol output by mfsymbol (mf,f).

```
? D = mfDelta(); mf = mfinit(D,0);
? PP = mfperiodpol(mf, D, -1); PP/=polcoef(PP, 1); bestappr(PP)
%1 = x^9 - 25/4*x^7 + 21/2*x^5 - 25/4*x^3 + x
? PM = mfperiodpol(mf, D, 1); PM/=polcoef(PM, 0); bestappr(PM)
%2 = -x^10 + 691/36*x^8 - 691/12*x^6 + 691/12*x^4 - 691/36*x^2 + 1
```

mfperiodpolbasis(k, flag)

Basis of period polynomials for weight k. If flag = 1 or -1, basis of odd or even period polynomials.

```
? mfperiodpolbasis(12,1)
%1 = [x^8 - 3*x^6 + 3*x^4 - x^2, x^10 - 1]
? mfperiodpolbasis(12,-1)
%2 = [4*x^9 - 25*x^7 + 42*x^5 - 25*x^3 + 4*x]
```

mfpetersson(fs, gs)

Petersson scalar product of the modular forms f and g belonging to the same modular form space mf, given by the corresponding "modular symbols" fs and gs output by mfsymbol (also in weight 1 and half-integral weight, where symbols do not exist). If gs is omitted it is understood to be equal to fs. The scalar product is normalized by the factor $1/[\Gamma : \Gamma_0(N)]$. Note that f and g can both be noncuspidal, in which case the program returns an error if the product is divergent. If the fields of definition $\mathbb{Q}(f)$ and $\mathbb{Q}(g)$ are equal to $\mathbb{Q}(\chi)$ the result is a scalar. If $[\mathbb{Q}(f) : \mathbb{Q}(\chi)] = d > 1$ and $[\mathbb{Q}(g) : \mathbb{Q}(\chi)] = e > 1$ the result is a *dxe* matrix corresponding to all the embeddings of f and g. In the intermediate cases d = 1 or e = 1 the result is a row or column vector.

```
? D=mfDelta(); mf=mfinit(D); DS=mfsymbol(mf,D); mfpetersson(DS)
%1 = 1.0353620568043209223478168122251645932 E-6
? mf=mfinit([11,6],0);B=mfeigenbasis(mf);BS=vector(#B,i,mfsymbol(mf,B[i]));
? mfpetersson(BS[1])
%3 = 1.6190120685220988139111708455305245466 E-5
? mfpetersson(BS[1],BS[2])
```

```
(continued from previous page)
```

```
%4 = [-3.826479006582967148 E-42 - 2.801547395385577002 E-41*I,\
    1.6661127341163336125 E-41 + 1.1734725972345985061 E-41*I,\
    0.E-42 - 6.352626992842664490 E-41*I]~
? mfpetersson(BS[2])
%5 =
[ 2.7576133733... E-5 2.0... E-42 6.3... E-43 ]
[ -4.1... E-42 6.77837030070... E-5 3.3...E-42 ]
[ -6.32...E-43 3.6... E-42 2.27268958069... E-5]
? mf=mfinit([23,2],0); F=mfeigenbasis(mf)[1]; FS=mfsymbol(mf,F);
? mfpetersson(FS)
%5 =
[0.0039488965740025031688548076498662860143 -3.56 ... E-40]
[ -3.5... E-40 0.0056442542987647835101583821368582485396]
```

Noncuspidal example:

```
? E1=mfeisenstein(5,1,-3);E2=mfeisenstein(5,-3,1);
? mf=mfinit([12,5,-3]); cusps=mfcusps(12);
? apply(x->mfcuspval(mf,E1,x),cusps)
%3 = [0, 0, 1, 0, 1, 1]
? apply(x->mfcuspval(mf,E2,x),cusps)
%4 = [1/3, 1/3, 0, 1/3, 0, 0]
? E1S=mfsymbol(mf,E1);E2S=mfsymbol(mf,E2);
? mfpetersson(E1S,E2S)
%6 = -1.884821671646... E-5 - 1.9... E-43*I
```

Weight 1 and 1/2-integral weight example:

```
? mf=mfinit([23,1,-23],1);F=mfbasis(mf)[1];FS=mfsymbol(mf,F);
? mfpetersson(mf,FS)
%2 = 0.035149946790370230814006345508484787443
? mf=mfinit([4,9/2],1);F=mfbasis(mf)[1];FS=mfsymbol(mf,F);
? mfpetersson(FS)
%4 = 0.00015577084407139192774373662467908966030
```

mfpow(F, n)

Compute F^n , where n is an integer and F is a generalized modular form:

```
? G = mfpow(mfEk(4), 3); \\ E4^3
? mfcoefs(G, 4)
%2 = [1, 720, 179280, 16954560, 396974160]
```

mfsearch(NK, V, space)

NK being of the form [N,k] with k possibly half-integral, search for a modular form with rational coefficients, of weight k and level N, whose initial coefficients a(0),... are equal to V; space specifies the modular form spaces in which to search, in mfinit or mfdim notation. The output is a list of matching forms with that given level and weight. Note that the character is of the form (D/.), where D is a (positive or negative) fundamental discriminant dividing N. The forms are sorted by increasing ||D||.

The parameter N can be replaced by a vector of allowed levels, in which case the list of forms is sorted by increasing level, then increasing ||D||. If a form is found at level N, any multiple of N with the same D is not considered. Some useful possibilities are

- [:math:`N_1..:math:N_2]`: all levels between N₁ and N₂, endpoints included;
- :math: F * [N₁..:math:N_2]: same but levels divisible by F;
- divisors (N_0) : all levels dividing N_0 .

Note that this is different from mfeigensearch, which only searches for rational eigenforms.

```
? F = mfsearch([[1..40], 2], [0,1,2,3,4], 1); #F
%1 = 3
? [ mfparams(f)[1..3] | f <- F ]
%2 = [[38, 2, 1], [40, 2, 8], [40, 2, 40]]
? mfcoefs(F[1],10)
%3 = [0, 1, 2, 3, 4, -5, -8, 1, -7, -5, 7]
```

mfshift(F, s)

Divide the generalized modular form F by q^s , omitting the remainder if there is one. One can have s < 0.

```
? D=mfDelta(); mfcoefs(mfshift(D,1), 4)
%1 = [1, -24, 252, -1472, 4830]
? mfcoefs(mfshift(D,2), 4)
%2 = [-24, 252, -1472, 4830, -6048]
? mfcoefs(mfshift(D,-1), 4)
%3 = [0, 0, 1, -24, 252]
```

mfshimura(mf, F, D)

F being a modular form of half-integral weight $k \ge 3/2$ and t a positive squarefree integer, returns the Shimura lift G of weight 2k - 1 corresponding to D. This function returns [mf2, G, v] where mf2 is a modular form space containing G and v expresses G in terms of mfbasis(mf2); so that G is mflinear(mf2, v).

```
? F = mfpow(mfTheta(), 7); mf = mfinit(F);
? [mf2, G, v] = mfshimura(mf, F, 3); mfcoefs(G,5)
\%2 = [-5/9, 280, 9240, 68320, 295960, 875280]
? mfparams(G) \\ the level may be lower than expected
\%3 = [1, 6, 1, y, t - 1]
? mfparams(mf2)
\%4 = [2, 6, 1, 4, t - 1]
? v
%5 = [280, 0]~
? mfcoefs(mf2, 5)
%6 =
[-1/504 - 1/504]
[ 1 0]
[ 33 1]
[ 244 0]
[ 1057 33]
```

[3126 0] ? mf = mfinit([60,5/2],1); F = mflinear(mf,mfkohnenbasis(mf)[,1]); ? mfparams(mfshimura(mf,F)[2]) %8 = [15, 4, 1, y, t - 1] ? mfparams(mfshimura(mf,F,6)[2]) %9 = [15, 4, 1, y, t - 1]

mfslashexpansion(*mf*, *f*, *g*, *n*, *flrat*, *params*, *precision*)

Let *mf* be a modular form space in level N, f a modular form belonging to *mf* and let g be in $M_2^+(Q)$. This function computes the Fourier expansion of $f||_k g$ to n terms. We first describe the behaviour when flrat is 0: the result is a vector v of floating point complex numbers such that

$$f \|_k g(\tau) = q^{\alpha} \sum_{m>=0} v[m+1]q^{m/w},$$

where $q = e(\tau)$, w is the width of the cusp g(ioo) (namely $(N/(c^2, N)$ if g is integral) and α is a rational number. If params is given, it is set to the parameters $[\alpha, w, matid(2)]$.

If flrat is 1, the program tries to rationalize the expression, i.e., to express the coefficients as rational numbers or polmods. We write $g = \lambda.M.A$ where $\lambda \in \mathbb{Q}^*$, $M \in SL_2(\mathbb{Z})$ and A = [a, b; 0, d] is upper triangular, integral and primitive with a > 0, d > 0 and $0 \le b < d$. Let α and w by the parameters attached to the expansion of $F := f \parallel_k M$ as above, i.e.

$$F(\tau) = q^{\alpha} \sum_{m > = 0} v[m+1]q^{m/w}.$$

The function returns the expansion v of $F = f \parallel_k M$ and sets the parameters to $[\alpha, w, A]$. Finally, the desired expansion is $(a/d)^{k/2}F(\tau + b/d)$. The latter is identical to the returned expansion when A is the identity, i.e. when $g \in PSL_2(\mathbb{Z})$. If this is not the case, the expansion differs from v by the multiplicative constant $(a/d)^{k/2}e(\alpha b/(dw))$ and a twist by a root of unity $q^{1/w} \to e(b/(dw))q^{1/w}$. The complications introduced by this extra matrix A allow to recognize the coefficients in a much smaller cyclotomic field, hence to obtain a simpler description overall. (Note that this rationalization step may result in an error if the program cannot perform it.)

```
? mf = mfinit([32,4],0); f = mfbasis(mf)[1];
? mfcoefs(f, 10)
\%2 = [0, 3, 0, 0, 0, 2, 0, 0, 0, 47, 0]
? mfatk = mfatkininit(mf,32); mfcoefs(mfatkin(mfatk,f),10) / mfatk[3]
\%3 = [0, 1, 0, 16, 0, 22, 0, 32, 0, -27, 0]
? mfatk[3] \ here normalizing constant C = 1, but need in general
\%4 = 1
? mfslashexpansion(mf,f,[0,-1;1,0],10,1,&params) * 32^(4/2)
\%5 = [0, 1, 0, 16, 0, 22, 0, 32, 0, -27, 0]
? params
\%6 = [0, 32, [1, 0; 0, 1]]
? mf = mfinit([12,8],0); f = mfbasis(mf)[1];
? mfslashexpansion(mf,f,[1,0;2,1],7,0)
%7 = [0, 0, 0, 0.66666666... + 0.E-38*I, 0, -3.9999999... + 6.92820...*I, 0,\
-11.999999999... - 20.78460969...*I]
? mfslashexpansion(mf,f,[1,0;2,1],7,1, &params)
%8 = [0, 0, 0, 2/3, 0, Mod(8*t, t^2+t+1), 0, Mod(-24*t-24, t^2+t+1)]
? params
\%9 = [0, 3, [1, 0; 0, 1]]
```

If $[\mathbb{Q}(f) : \mathbb{Q}(\chi)] > 1$, the coefficients may be polynomials in y, where y is any root of the polynomial giving the field of definition of f (f.mod or mfparams(f)[4]).

? mf=mfinit([23,2],0);f=mfeigenbasis(mf)[1]; ? mfcoefs(f,5) %1 = [Mod(0, y² - y - 1), Mod(1, y² - y - 1), Mod(-y, y² - y - 1), Mod(2*y - 1, y² - y - 1), Mod(y - 1, y² - y - 1), Mod(-2*y, y² - y - 1)] ? mfslashexpansion(mf,f,[1,0;0,1],5,1) %2 = [0, 1, -y, 2*y - 1, y - 1, -2*y] ? mfslashexpansion(mf,f,[0,-1;1,0],5,1) %3 = [0, -1/23, 1/23*y, -2/23*y + 1/23, -1/23*y + 1/23, 2/23*y]

Caveat. In half-integral weight, we define the "slash" operation as

$$(f||_k g)(\tau) := ((c\tau + d)^{-1/2})^{2k} f(g.\tau),$$

with the principal determination of the square root. In particular, the standard cocycle condition is no longer satisfied and we only have f ||(gg') = (f||g)||g'.

mfspace(mf, f)

Identify the modular space mf, resp. the modular form f in mf if present, as the flag given to mfinit. Returns 0 (newspace), 1 (cuspidal space), 2 (old space), 3 (Eisenstein space) or 4 (full space).

```
? mf = mfinit([1,12],1); mfspace(mf)
%1 = 1
? mfspace(mf, mfDelta())
%2 = 0 \\ new space
```

This function returns -1 when the form f is modular but does not belong to the space.

? mf = mfinit([1,2]; mfspace(mf, mfEk(2))
%3 = -1

When f is not modular and is for instance only quasi-modular, the function returns nonsense:

```
? M6 = mfinit([1,6]);
? dE4 = mfderiv(mfEk(4)); \\ not modular !
? mfspace(M6,dE4) \\ asserts (wrongly) that E4' belongs to new space
%3 = 0
```

mfsplit(mf, dimlim, flag)

mf from mfinit with integral weight containing the new space (either the new space itself or the cuspidal space or the full space), and preferably the newspace itself for efficiency, split the space into Galois orbits of eigenforms of the newspace, satisfying various restrictions.

The functions returns [vF, vK], where vF gives (Galois orbit of) eigenforms and vK is a list of polynomials defining each Galois orbit. The eigenforms are given in mftobasis format, i.e. as a matrix whose columns give the forms with respect to mfbasis(mf).

If dimlim is set, only the Galois orbits of dimension $\langle = dimlim$ are computed (i.e. the rational eigenforms if dimlim = 1 and the character is real). This can considerably speed up the function when a Galois orbit is defined over a large field.

flag speeds up computations when the dimension is large: if flag = d > 0, when the dimension of the eigenspace is > d, only the Galois polynomial is computed.

Note that the function mfeigenbasis returns all eigenforms in an easier to use format (as modular forms which can be input as is in other functions); mfsplit is only useful when you can restrict to orbits of small dimensions, e.g. rational eigenforms.

```
? mf=mfinit([11,2],0); f=mfeigenbasis(mf)[1]; mfcoefs(f,16)
\%1 = [0, 1, -2, -1, \ldots]
? mf=mfinit([23,2],0); f=mfeigenbasis(mf)[1]; mfcoefs(f,16)
%2 = [Mod(0, z^2 - z - 1), Mod(1, z^2 - z - 1), Mod(-z, z^2 - z - 1), ...]
? mf=mfinit([179,2],0); apply(poldegree, mffields(mf))
\%3 = [1, 3, 11]
? mf=mfinit([719,2],0);
? [vF,vK] = mfsplit(mf, 5); \\ fast when restricting to small orbits
time = 192 \text{ ms.}
? #vF \setminus a single orbit
\%5 = 1
? poldegree(vK[1]) \\ of dimension 5
\%6 = 5
? [vF,vK] = mfsplit(mf); \\ general case is slow
time = 2,104 ms.
? apply(poldegree,vK)
\%8 = [5, 10, 45] \setminus because degree 45 is large...
```

mfsturm(NK)

Gives the Sturm bound for modular forms on $\Gamma_0(N)$ and weight k, i.e., an upper bound for the order of the zero at infinity of a nonzero form. NK is either

- a pair [N, k], in which case the bound is the floor of (kN/12). $\prod_{p \parallel N} (1 + 1/p)$;
- or the output of mfinit in which case the exact upper bound is returned.

```
? NK = [96,6]; mfsturm(NK)
%1 = 97
? mf=mfinit(NK,1); mfsturm(mf)
%2 = 76
? mfdim(NK,0) \\ new space
%3 = 72
```

mfsymbol(*mf*, *f*, *precision*)

Initialize data for working with all period polynomials of the modular form f: this is essential for efficiency for functions such as mfsymboleval, mfmanin, and mfpetersson. An mfsymbol contains an mf structure and can always be used whenever an mf would be needed.

```
? mf=mfinit([23,2],0);F=mfeigenbasis(mf)[1];
? FS=mfsymbol(mf,F);
? mfsymboleval(FS,[0,oo])
%3 = [8.762565143790690142 E-39 + 0.0877907874...*I,
-5.617375463602574564 E-39 + 0.0716801031...*I]
? mfpetersson(FS)
%4 =
[0.0039488965740025031688548076498662860143 1.2789721111175127425 E-40]
[1.2630501762985554269 E-40 0.0056442542987647835101583821368582485396]
```

By abuse of language, initialize data for working with mfpetersson in weight 1 and half-integral weight (where no symbol exist); the mf argument may be an mfsymbol attached to a form on the space, which avoids recomputing

data independent of the form.

```
? mf=mfinit([12,9/2],1); F=mfbasis(mf);
? fs=mfsymbol(mf,F[1]);
time = 476 ms
? mfpetersson(fs)
%2 = 1.9722437519492014682047692073275406145 E-5
? f2s = mfsymbol(mf,F[2]);
time = 484 ms.
? mfpetersson(f2s)
%4 = 1.2142222531326333658647877864573002476 E-5
? gs = mfsymbol(fs,F[2]); \\ re-use existing symbol, a little faster
time = 430 ms.
? mfpetersson(gs) == %4 \\ same value
%6 = 1
```

For simplicity, we also allow mfsymbol(f) instead of mfsymbol(mfinit(f), f):

mfsymboleval(*fs*, *path*, *ga*, *precision*)

Evaluation of the modular symbol fs (corresponding to the modular form f) output by mfsymbol on the given path path, where path is either a vector $[s_1, s_2]$ or an integral matrix [a, b; c, d] representing the path [a/c, b/d]. In both cases s_1 or s_2 (or a/c or b/d) can also be elements of the upper half-plane. To avoid possibly lengthy mfsymbol computations, the program also accepts fs of the form [mf, F], but in that case s_1 and s_2 are limited to oo and elements of the upper half-plane. The result is the polynomial equal to $\int_{s_1}^{s_2} (X - \tau)^{k-2} F(\tau) d\tau$, the integral being computed along a geodesic joining s_1 and s_2 . If ga in $GL_2^+(\mathbb{Q})$ is given, replace F by $F||_k \gamma$. Note that if the integral diverges, the result will be a rational function. If the field of definition $\mathbb{Q}(f)$ is larger than $\mathbb{Q}(\chi)$ then f can be embedded into \mathbb{C} in $d = [\mathbb{Q}(f) : \mathbb{Q}(\chi)]$ ways, in which case a vector of the d results is returned.

```
? mf=mfinit([35,2],1);f=mfbasis(mf)[1];fs=mfsymbol(mf,f);
? mfsymboleval(fs,[0,oo])
%1 = 0.31404011074188471664161704390256378537*I
? mfsymboleval(fs,[1,3;2,5])
%2 = -0.1429696291... - 0.2619975641...*I
? mfsymboleval(fs,[I,2*I])
%3 = 0.00088969563028739893631700037491116258378*I
? E2=mfEk(2);E22=mflinear([E2,mfbd(E2,2)],[1,-2]);mf=mfinit(E22);
? E2S = mfsymbol(mf,E22);
? mfsymboleval(E2S,[0,1])
%6 = (-1.00000...*x^2 + 1.00000...*x - 0.50000...)/(x^2 - x)
```

The rational function which is given in case the integral diverges is easy to interpret. For instance:

```
? E4=mfEk(4);mf=mfinit(E4);ES=mfsymbol(mf,E4);
? mfsymboleval(ES,[I,oo])
%2 = 1/3*x^3 - 0.928067...*I*x^2 - 0.833333...*x + 0.234978...*I
? mfsymboleval(ES,[0,I])
%3 = (-0.234978...*I*x^3 - 0.833333...*x^2 + 0.928067...*I*x + 0.333333...)/x
```

mfsymboleval(ES,[a,oo]) is the limit as $T \rightarrow oo$ of

$$\int_{a}^{iT} (X-\tau)^{k-2} F(\tau) d\tau + a(0)(X-iT)^{k-1}/(k-1)$$

where a(0) is the 0 at infinity. Similarly, mfsymboleval(ES,[0,a]) is the limit as $T \to oo$ of

$$\int_{i/T}^{a} (X-\tau)^{k-2} F(\tau) d\tau + b(0)(1+iTX)^{k-1}/(k-1),$$

where b(0) is the 0 at infinity.

mftaylor(*F*, *n*, *flreal*, *precision*)

F being a form in $M_k(SL_2(\mathbb{Z}))$, computes the first n + 1 canonical Taylor expansion of F around $\tau = I$. If flreal = 0, computes only an algebraic equivalence class. If flreal is set, compute p_n such that for τ close enough to I we have

$$f(\tau) = (2I/(\tau+I))^k \sum_{n>=0} p_n ((\tau-I)/(\tau+I))^n.$$

? D=mfDelta(); ? mftaylor(D,8) %2 = [1/1728, 0, -1/20736, 0, 1/165888, 0, 1/497664, 0, -11/3981312]

mftobasis(mf, F, flag)

Coefficients of the form F on the basis given by mfbasis(mf). A q-expansion or vector of coefficients can also be given instead of F, but in this case an error message may occur if the expansion is too short. An error message is also given if F does not belong to the modular form space. If flag is set, instead of error messages the output is an affine space of solutions if a q-expansion or vector of coefficients is given, or the empty column otherwise.

? mf = mfinit([26,2],0); mfdim(mf) %1 = 2 ? F = mflinear(mf,[a,b]); mftobasis(mf,F) %2 = [a, b]~

A q-expansion or vector of coefficients can also be given instead of F.

```
? Th = 1 + 2*sum(n=1, 8, q^(n^2), 0(q^80));
? mf = mfinit([4,5,Mod(3,4)]);
? mftobasis(mf, Th^10)
%3 = [64/5, 4/5, 32/5]~
```

If F does not belong to the corresponding space, the result is incorrect and simply matches the coefficients of F up to some bound, and the function may either return an empty column or an error message. If flag is set, there are no error messages, and the result is an empty column if F is a modular form; if F is supplied via a series or vector of coefficients which does not contain enough information to force a unique (potential) solution, the function returns [v, K] where v is a solution and K is a matrix of maximal rank describing the affine space of potential solutions v + K.x.

```
? mftobasis(mf, mfEk(10))
 *** at top-level: mftobasis(mf,mfEk(
 *** ^------
 *** mftobasis: domain error in mftobasis: form does not belong to space
? mftobasis(mf, mfEk(10), 1)
%7 = []~
```

mftocoset(N, M, Lcosets)

M being a matrix in $SL_2(Z)$ and Lcosets being mfcosets (N), a list of right cosets of $\Gamma_0(N)$, find the coset to which M belongs. The output is a pair $[\gamma, i]$ such that $M = \gamma Lcosets[i], \gamma \in \Gamma_0(N)$.

? N = 4; L = mfcosets(N); ? mftocoset(N, [1,1;2,3], L) %2 = [[-1, 1; -4, 3], 5]

mftonew(mf, F)

mf being being a full or cuspidal space with parameters $[N, k, \chi]$ and F a cusp form in that space, returns a vector of 3-component vectors [M, d, G], where $f(\chi) ||M||N, d||N/M$, and G is a form in $S_k^{new}(\Gamma_0(M), \chi)$ such that F is equal to the sum of the B(d)(G) over all these 3-component vectors.

? mf = mfinit([96,6],1); F = mfbasis(mf)[60]; s = mftonew(mf,F); #s %1 = 1 ? [M,d,G] = s[1]; [M,d] %2 = [48, 2] ? mfcoefs(F,10) %3 = [0, 0, -160, 0, 0, 0, 0, 0, 0, 0, -14400] ? mfcoefs(G,10) %4 = [0, 0, -160, 0, 0, 0, 0, 0, 0, 0, -14400]

mftraceform(NK, space)

If NK = [N, k, CHI, .] as in mfinit with k integral, gives the trace form in the corresponding subspace of $S_k(\Gamma_0(N), \chi)$. The supported values for space are 0: the newspace (default), 1: the full cuspidal space.

```
? F = mftraceform([23,2]); mfcoefs(F,16)
%1 = [0, 2, -1, 0, -1, -2, -5, 2, 0, 4, 6, -6, 5, 6, 4, -10, -3]
? F = mftraceform([23,1,-23]); mfcoefs(F,16)
%2 = [0, 1, -1, -1, 0, 0, 1, 0, 1, 0, 0, 0, 0, -1, 0, 0, -1]
```

mftwist(F, D)

F being a generalized modular form, returns the twist of F by the integer D, i.e., the form G such that mfcoef(G, n) = ::math::(D/n)mfcoef(F,n), where (D/n) is the Kronecker symbol.

```
? mf = mfinit([11,2],0); F = mfbasis(mf)[1]; mfcoefs(F, 5)
%1 = [0, 1, -2, -1, 2, 1]
? G = mftwist(F,-3); mfcoefs(G, 5)
%2 = [0, 1, 2, 0, 2, -1]
? mf2 = mfinit([99,2],0); mftobasis(mf2, G)
%3 = [1/3, 0, 1/3, 0]~
```

Note that twisting multiplies the level by D^2 . In particular it is not an involution:

? H = mftwist(G,-3); mfcoefs(H, 5) %4 = [0, 1, -2, 0, 2, 1] ? mfparams(G) %5 = [99, 2, 1, y, t - 1]

 $\min(x, y)$

Creates the maximum of x and y when they can be compared.

minpoly(A, v)

minimal polynomial of A with respect to the variable v., i.e. the monic polynomial P of minimal degree (in the variable v) such that P(A) = 0.

modreverse(z)

Let z = Mod(A, T) be a polmod, and Q be its minimal polynomial, which must satisfy deg(Q) = deg(T). Returns a "reverse polmod" Mod(B, Q), which is a root of T.

This is quite useful when one changes the generating element in algebraic extensions:

? u = Mod(x, x³ - x -1); v = u⁵; ? w = modreverse(v) %2 = Mod(x² - 4^{*}x + 1, x³ - 5^{*}x² + 4^{*}x - 1)

which means that $x^3 - 5x^2 + 4x - 1$ is another defining polynomial for the cubic field

$$\mathbb{Q}(u) = \mathbb{Q}[x]/(x^3 - x - 1) = \mathbb{Q}[x]/(x^3 - 5x^2 + 4x - 1) = \mathbb{Q}(v),$$

and that $u \to v^2 - 4v + 1$ gives an explicit isomorphism. From this, it is easy to convert elements between the $A(u) \in \mathbb{Q}(u)$ and $B(v) \in \mathbb{Q}(v)$ representations:

? A = u^2 + 2*u + 3; subst(lift(A), 'x, w) %3 = Mod(x^2 - 3*x + 3, x^3 - 5*x^2 + 4*x - 1) ? B = v^2 + v + 1; subst(lift(B), 'x, v) %4 = Mod(26*x^2 + 31*x + 26, x^3 - x - 1)

If the minimal polynomial of z has lower degree than expected, the routine fails

```
? u = Mod(-x^3 + 9*x, x^4 - 10*x^2 + 1)
? modreverse(u)
*** modreverse: domain error in modreverse: deg(minpoly(z)) < 4
*** Break loop: type 'break' to go back to GP prompt
break> Vec( dbg_err() ) \\ ask for more info
["e_DOMAIN", "modreverse", "deg(minpoly(z))", "<", 4,
Mod(-x^3 + 9*x, x^4 - 10*x^2 + 1)]
break> minpoly(u)
x^2 - 8
```

moebius(x)

Moebius μ -function of ||x||; x must be a nonzero integer.

msatkinlehner(M, Q, H)

Let M be a full modular symbol space of level N, as given by msinit, let Q||N, (Q, N/Q) = 1, and let H be a subspace stable under the Atkin-Lehner involution w_Q . Return the matrix of w_Q acting on H (M if omitted).

```
? M = msinit(36,2); \\ M_2(Gamma_0(36))
? w = msatkinlehner(M,4); w^2 == 1
%2 = 1
? #w \\ involution acts on a 13-dimensional space
%3 = 13
? M = msinit(36,2, -1); \\ M_2(Gamma_0(36))^-
? w = msatkinlehner(M,4); w^2 == 1
%5 = 1
? #w
%6 = 4
```

mscosets(gen, inH)

gen being a system of generators for a group G and H being a subgroup of finite index in G, return a list of right cosets of H and the right action of G on H. The subgroup H is given by a criterion inH (closure) deciding whether an element of G belongs to H. The group G is restricted to types handled by generic multiplication (*) and inversion (g^(-1)), such as matrix groups or permutation groups.

Let $gens = [g_1, ..., g_r]$. The function returns [C, M] where C lists the h = [G : H] representatives $[\gamma_1, ..., \gamma_h]$ for the right cosets $H\gamma_1, ..., H\gamma_h$; γ_1 is always the neutral element in G. For all $i \le h, j \le r$, if M[i][j] = k then $H\gamma_ig_j = H\gamma_k$.

```
? PSL2 = [[0,1;-1,0], [1,1;0,1]]; \\ S and T
\\ G = PSL2, H = Gamma0(2)
? [C, M] = mscosets(PSL2, g->g[2,1] % 2 == 0);
? C \\ three cosets
%3 = [[1, 0; 0, 1], [0, 1; -1, 0], [0, 1; -1, -1]]
? M
%4 = [Vecsmall([2, 1]), Vecsmall([1, 3]), Vecsmall([3, 2])]
```

Looking at M[1] we see that S belongs to the second coset and T to the first (trivial) coset.

mscuspidal(M, flag)

M being a full modular symbol space, as given by msinit, return its cuspidal part S. If flag = 1, return [S, E] its decomposition into cuspidal and Eisenstein parts.

A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

```
? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+
? [S,E] = mscuspidal(M, 1);
? E[1] \\ 2-dimensional
%3 =
[0 -10]
[0 -15]
[0 -3]
[1 0]
? S[1] \\ 1-dimensional
%4 =
[ 3]
```

[30]		
[6]		
[-8]		

msdim(M)

M being a full modular symbol space or subspace, for instance as given by msinit or mscuspidal, return its dimension as a \mathbb{Q} -vector space.

```
? M = msinit(11,4); msdim(M)
%1 = 6
? M = msinit(11,4,1); msdim(M)
%2 = 4 \\ dimension of the '+' part
? [S,E] = mscuspidal(M,1);
? [msdim(S), msdim(E)]
%4 = [2, 2]
```

Note that mfdim([N,k]) is going to be much faster if you only need the dimension of the space and not really to work with it. This function is only useful to quickly check the dimension of an existing space.

mseisenstein(M)

M being a full modular symbol space, as given by msinit, return its Eisenstein subspace. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace. This is the same basis as given by the second component of mscuspidal(M, 1).

```
? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+
? E = mseisenstein(M);
? E[1] \\ 2-dimensional
%3 =
[0 -10]
[0 -15]
[0 -15]
[0 -3]
[1 0]
? E == mscuspidal(M,1)[2]
%4 = 1
```

mseval(M, s, p)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M be a full modular symbol space, as given by msinit, let s be a modular symbol from M, i.e. an element of $\operatorname{Hom}_G(\Delta_0, V)$, and let $p = [a, b] \in \Delta_0$ be a path between two elements in $\mathbb{P}^1(\mathbb{Q})$, return $s(p) \in V$. The path extremities a and b may be given as t_INT, t_FRAC or oo = (1:0); it is also possible to describe the path by a $2x^2$ integral matrix whose columns give the two cusps. The symbol s is either

- a t_COL coding a modular symbol in terms of the fixed basis of Hom_G(Δ₀, V) chosen in M; if M was initialized with a nonzero sign (+ or −), then either the basis for the full symbol space or the -part can be used (the dimension being used to distinguish the two).
- a t_MAT whose columns encode modular symbols as above. This is much faster than evaluating individual symbols on the same path p independently.
• a t_VEC (v_i) of elements of V, where the $v_i = s(g_i)$ give the image of the generators g_i of Δ_0 , see mspathgens. We assume that s is a proper symbol, i.e. that the v_i satisfy the mspathgens relations.

If p is omitted, convert a single symbol s to the second form: a vector of the $s(g_i)$. A t_MAT is converted to a vector of such.

```
? M = msinit(2,8,1); \\ M_8(Gamma_0(2))^+
? g = mspathgens(M)[1]
%2 = [[+oo, 0], [0, 1]]
? N = msnew(M)[1]; #N \\ Q-basis of new subspace, dimension 1
%3 = 1
? s = N[,1] \\ t_COL representation
%4 = [-3, 6, -8]~
? S = mseval(M, s) \\ t_VEC representation
%5 = [64*x^6-272*x^4+136*x^2-8, 384*x^5+960*x^4+192*x^3-672*x^2-432*x-72]
? mseval(M,s, g[1])
%6 = 64*x^6 - 272*x^4 + 136*x^2 - 8
? mseval(M,S, g[1])
%7 = 64*x^6 - 272*x^4 + 136*x^2 - 8
```

Note that the symbol should have values in $V = \mathbb{Q}[x, y]_{k-2}$, we return the de-homogenized values corresponding to y = 1 instead.

msfarey(*F*, *inH*, *CM*)

F being a Farey symbol attached to a group G contained in $PSL_2(\mathbb{Z})$ and H a subgroup of G, return a Farey symbol attached to H. The subgroup H is given by a criterion inH (closure) deciding whether an element of G belongs to H. The symbol F can be created using

- mspolygon: $G = \Gamma_0(N)$, which runs in time O(N);
- or msfarey itself, which runs in time $O([G:H]^2)$.

If present, the argument CM is set to mscosets(F[3]), giving the right cosets of H and the action of G by right multiplication. Since msfarey's algorithm is quadratic in the index [G : H], it is advisable to construct subgroups by a chain of inclusions if possible.

```
\ \ Gamma_0(N)
GO(N) = mspolygon(N);
\\ Gamma_1(N): direct construction, slow
G1(N) = msfarey(mspolygon(1), g \rightarrow my(a = g[1,1]%N, c = g[2,1]%N);
c == 0 \&\& (a == 1 || a == N-1));
\\ Gamma_1(N) via Gamma_0(N): much faster
G1(N) = msfarey(GO(N), g \rightarrow my(a=g[1,1]%N); a==1 || a==N-1);
\ \ Gamma(N)
G(N) = msfarey(G1(N), g \rightarrow g[1,2]%N==0);
G_{00}(N) = msfarey(GO(N), x \rightarrow x[1,2]%N==0);
G1_0(N1,N2) = msfarey(G0(1), x \rightarrow x[2,1]%N1==0 \& x[1,2]%N2==0);
\ Gamma_0(91) has 4 elliptic points of order 3, Gamma_1(91) has none
D0 = mspolygon(G0(91), 2)[4];
D1 = mspolygon(G1(91), 2)[4];
write("F.tex","\\documentclass{article}\\usepackage{tikz}\\begin{document}",\
D0, "\n", D1, "\\end{document}");
```

msfromcusp(M, c)

Returns the modular symbol attached to the cusp c, where M is a modular symbol space of level N, attached to $G = \Gamma_0(N)$. The cusp c in $\mathbb{P}^1(\mathbb{Q})/G$ is given either as oo (=(1:0)) or as a rational number a/b (=(a:b)). The attached symbol maps the path $[b] - [a] \in Div^0(\mathbb{P}^1(\mathbb{Q}))$ to $E_c(b) - E_c(a)$, where $E_c(r)$ is 0 when r! = c and $X^{k-2} ||_{\gamma_r}$ otherwise, where $\gamma_r \cdot r = (1:0)$. These symbols span the Eisenstein subspace of M.

```
? M = msinit(2,8); \setminus M_8(Gamma_0(2))
? E = mseisenstein(M);
? E[1] \\ two-dimensional
%3 =
[0 - 10]
[0 - 15]
[0 - 3]
[1 0]
? s = msfromcusp(M,oo)
\%4 = [0, 0, 0, 1] \sim
? mseval(M, s)
\%5 = [1, 0]
? s = msfromcusp(M,1)
\%6 = [-5/16, -15/32, -3/32, 0] \sim
? mseval(M,s)
\%7 = [-x^{6}, -6^{*}x^{5} - 15^{*}x^{4} - 20^{*}x^{3} - 15^{*}x^{2} - 6^{*}x - 1]
```

In case M was initialized with a nonzero *sign*, the symbol is given in terms of the fixed basis of the whole symbol space, not the + or - part (to which it need not belong).

? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+ ? E = mseisenstein(M); ? E[1] \\ still two-dimensional, in a smaller space %3 = [0 -10] [0 3] [-1 0] ? s = msfromcusp(M,oo) \\ in terms of the basis for M_8(Gamma_0(2)) ! %4 = [0, 0, 0, 1]~ ? mseval(M, s) \\ same symbol as before %5 = [1, 0]

msfromell(E, sign)

Let E/\mathbb{Q} be an elliptic curve of conductor N. For $\varepsilon = 1$, we define the (cuspidal, new) modular symbol x^{ε} in $H^1_c(X_0(N), \mathbb{Q})^{\varepsilon}$ attached to E. For all primes p not dividing N we have $T_p(x^{\varepsilon}) = a_p x^{\varepsilon}$, where $a_p = p + 1 - \#E(\mathbb{F}_p)$.

Let $\Omega^+ = E.omega[1]$ be the real period of E (integration of the Néron differential $dx/(2y + a_1x + a_3)$ on the connected component of $E(\mathbb{R})$, i.e. the generator of $H_1(E, \mathbb{Z})^+$) normalized by $\Omega^+ > 0$. Let $i\Omega^-$ the integral on a generator of $H_1(E, \mathbb{Z})^-$ with $\Omega^- \in \mathbb{R}_{>0}$. If $c_o o$ is the number of connected components of $E(\mathbb{R})$, Ω^- is

equal to $(-2/c_o o)ximag(E.omega[2])$. The complex modular symbol is defined by

$$F:\delta\to 2i\pi\int_{\delta}f(z)dz$$

The modular symbols x^{ε} are normalized so that $F = x^{+}\Omega^{+} + x^{-}i\Omega^{-}$. In particular, we have

$$x^+([0] - [oo]) = L(E, 1)/\Omega^+$$

which defines x unless L(E, 1) = 0. Furthermore, for all fundamental discriminants D such that $\varepsilon D > 0$, we also have

$$\sum_{0 < a < \|D\|} (D\|a) x^{\varepsilon} ([a/\|D\|] - [oo]) = L(E, (D\|.), 1) / \Omega^{\varepsilon},$$

where (D||.) is the Kronecker symbol. The period Ω^- is also $2/c_o ox$ the real period of the twist $E^{(-4)} = elltwist(E, -4)$.

This function returns the pair [M, x], where M is msinit(N, 2) and x is x^{sign} as above when sign = 1, and $x = [x^+, x^-, L_E]$ when sign is 0, where L_E is a matrix giving the canonical \mathbb{Z} -lattice attached to E in the sense of mslattice applied to $\mathbb{Q}x^+ + \mathbb{Q}x^-$. Explicitly, it is generated by (x^+, x^-) when $E(\mathbb{R})$ has two connected components and by $(x^+ - x^-, 2x^-)$ otherwise.

The modular symbols x are given as a t_COL (in terms of the fixed basis of $\text{Hom}_G(\Delta_0, \mathbb{Q})$ chosen in M).

? E=ellinit([0,-1,1,-10,-20]); \\ X_0(11) ? [M,xp]= msfromell(E,1); ? xp %3 = [1/5, -1/2, -1/2]~ ? [M,x]= msfromell(E); ? x \\ x^+, x^- and L_E %5 = [[1/5, -1/2, -1/2]~, [0, 1/2, -1/2]~, [1/5, 0; -1, 1; 0, -1]] ? p = 23; (mshecke(M,p) - ellap(E,p))*x[1] %6 = [0, 0, 0]~ \\ true at all primes, including p = 11; same for x[2] ? (mshecke(M,p) - ellap(E,p))*x[3] == 0 %7 = 1

Instead of a single curve E, one may use instead a vector of *isogenous* curves. The function then returns M and the vector of attached modular symbols.

msfromhecke(M, v, H)

Given a mainit M and a vector v of pairs [p, P] (where p is prime and P is a polynomial with integer coefficients), return a basis of all modular symbols such that $P(T_p)(s) = 0$. If H is present, it must be a Hecke-stable subspace and we restrict to $s \in H$. When T_p has a rational eigenvalue and $P(x) = x - a_p$ has degree 1, we also accept the integer a_p instead of P.

```
? E = ellinit([0,-1,1,-10,-20]) \\11a1
? ellap(E,2)
%2 = -2
? ellap(E,3)
%3 = -1
? M = msinit(11,2);
? S = msfromhecke(M, [[2,-2],[3,-1]])
%5 =
[ 1 1]
```

```
[-5 0]
[ 0 -5]
? mshecke(M, 2, S)
%6 =
[-2 0]
[ 0 -2]
? M = msinit(23,4);
? S = msfromhecke(M, [[5, x^4-14*x^3-244*x^2+4832*x-19904]]);
? factor( charpoly(mshecke(M,5,S)) )
%9 =
[x^4 - 14*x^3 - 244*x^2 + 4832*x - 19904 2]
```

msgetlevel(M)

M being a full modular symbol space, as given by msinit, return its level N.

msgetsign(M)

M being a full modular symbol space, as given by msinit, return its sign: 1 or 0 (unset).

```
? M = msinit(11,4, 1);
? msgetsign(M)
%2 = 1
? M = msinit(11,4);
? msgetsign(M)
%4 = 0
```

msgetweight(M)

M being a full modular symbol space, as given by msinit, return its weight k.

```
? M = msinit(11,4);
? msgetweight(M)
%2 = 4
```

mshecke(M, p, H)

M being a full modular symbol space, as given by msinit, p being a prime number, and H being a Hecke-stable subspace (M if omitted) return the matrix of T_p acting on H (U_p if p divides N). Result is undefined if H is not stable by T_p (resp. U_p).

```
? M = msinit(11,2); \\ M_2(Gamma_0(11))
? T2 = mshecke(M,2)
%2 =
[3 0 0]
[1 -2 0]
[1 0 -2]
? M = msinit(11,2, 1); \\ M_2(Gamma_0(11))^+
? T2 = mshecke(M,2)
%4 =
[ 3 0]
```

```
[-1 -2]
? N = msnew(M)[1] \\ Q-basis of new cuspidal subspace
%5 =
[-2]
[-5]
? p = 1009; mshecke(M, p, N) \\ action of T_1009 on N
%6 =
[-10]
? ellap(ellinit("11a1"), p)
%7 = -10
```

msinit(G, V, sign)

Given G a finite index subgroup of $SL(2,\mathbb{Z})$ and a finite dimensional representation V of $GL(2,\mathbb{Q})$, creates a space of modular symbols, the G-module $\operatorname{Hom}_G(Div^0(\mathbb{P}^1(\mathbb{Q})), V)$. This is canonically isomorphic to $H^1_c(X(G), V)$, and allows to compute modular forms for G. If sign is present and nonzero, it must be 1 and we consider the subspace defined by $Ker(\sigma - sign)$, where σ is induced by [-1,0;0,1]. Currently the only supported groups are the $\Gamma_0(N)$, coded by the integer N > 0. The only supported representation is $V_k = \mathbb{Q}[X, Y]_{k-2}$, coded by the integer $k \ge 2$.

```
? M = msinit(11,2); msdim(M) \\ Gamma0(11), weight 2
\%1 = 3
? mshecke(M,2) \setminus T_2 acting on M
%2 =
[3 1 1]
[0 -2 0]
[0 0 -2]
? msstar(M) \setminus \ * \ involution
%3 =
[1 \ 0 \ 0]
[0 0 1]
[0 1 0]
? Mp = msinit(11,2, 1); msdim(Mp) \\ + part
\%4 = 2
? mshecke(Mp,2) \setminus T_2 action on M^+
%5 =
[3 2]
[0 - 2]
? msstar(Mp)
%6 =
[1 0]
[0 1]
```

msissymbol(M, s)

M being a full modular symbol space, as given by msinit, check whether s is a modular symbol attached to M. If A is a matrix, check whether its columns represent modular symbols and return a 0 - 1 vector.

```
? M = msinit(7,8, 1); \\ M_8(Gamma_0(7))^+
? A = msnew(M)[1];
? s = A[,1];
? msissymbol(M, s)
\%4 = 1
? msissymbol(M, A)
\%5 = [1, 1, 1]
? S = mseval(M,s);
? msissymbol(M, S)
%7 = 1
? [g,R] = mspathgens(M); g
%8 = [[+oo, 0], [0, 1/2], [1/2, 1]]
? #R \ 3 relations among the generators g_i
%9 = 3
? T = S; T[3] ++; \\ randomly perturb S(g_3)
? msissymbol(M, T)
%11 = 0 \setminus no longer satisfies the relations
```

mslattice(M, H)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$ and $V_k = \mathbb{Q}[x, y]_{k-2}$. Let M be a full modular symbol space, as given by msinit and let H be a subspace, e.g. as given by mscuspidal. This function returns a canonical \mathbb{Z} structure on H defined as follows. Consider the map $c : M = \operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, V_k) \to H^1(\Gamma_0(N), V_k)$ given by $\phi : -- > class(\gamma \to \phi(0, \gamma^{-1}0))$. Let $L_k = \mathbb{Z}[x, y]_{k-2}$ be the natural \mathbb{Z} -structure of V_k . The result of mslattice is a \mathbb{Z} -basis of the inverse image by c of $H^1(\Gamma_0(N), L_k)$ in the space of modular symbols generated by H.

For user convenience, H can be defined by a matrix representing the \mathbb{Q} -basis of H (in terms of the canonical \mathbb{Q} -basis of M fixed by msinit and used to represent modular symbols).

If omitted, H is the cuspidal part of M as given by mscuspidal. The Eisenstein part $\operatorname{Hom}_{\Gamma_0(N)}(Div(\mathbb{P}^1(\mathbb{Q})), V_k)$ is in the kernel of c, so the result has no meaning for the Eisenstein part H.

```
? M=msinit(11,2);
? [S,E] = mscuspidal(M,1); S[1] \\ a primitive Q-basis of S
%2 =
[ 1 1]
[-5 0]
[0 -5]
? mslattice(M,S)
%3 =
[-1/5 - 1/5]
[ 1 0]
[01]
? mslattice(M,E)
%4 =
[1]
[0]
[0]
? M=msinit(5,4);
? S=mscuspidal(M); S[1]
```

%6 =
[7 20]
[3 3]
[-10 -23]
[-30 -30]
? mslattice(M,S)
%7 =
[-1/10 -11/130]
[0 -1/130]
[1/10 6/65]
[0 1/13]

msnew(M)

M being a full modular symbol space, as given by msinit, return the *new* part of its cuspidal subspace. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

```
? M = msinit(11,8, 1); \\ M_8(Gamma_0(11))^+
? N = msnew(M);
? #N[1] \\ 6-dimensional
%3 = 6
```

msomseval(Mp, PHI, path)

Return the vectors of moments of the *p*-adic distribution attached to the path path by the overconvergent modular symbol PHI.

? M = msinit(3,6,1); ? Mp= mspadicinit(M,5,10); ? phi = [5,-3,-1]~; ? msissymbol(M,phi) %4 = 1 ? PHI = mstooms(Mp,phi); ? ME = msomseval(Mp,PHI,[oo, 0]);

mspadicL(mu, s, r)

Returns the value (or r-th derivative) on a character χ^s of \mathbb{Z}_p^* of the p-adic L-function attached to mu.

Let Φ be the *p*-adic distribution-valued overconvergent symbol attached to a modular symbol ϕ for $\Gamma_0(N)$ (eigenvector for $T_N(p)$ for the eigenvalue a_p). Then $L_p(\Phi, \chi^s) = L_p(\mu, s)$ is the *p*-adic *L* function defined by

$$L_p(\Phi, \chi^s) = \int_{\mathbb{Z}_p^*} \chi^s(z) d\mu(z)$$

where μ is the distribution on \mathbb{Z}_p^* defined by the restriction of $\Phi([oo] - [0])$ to \mathbb{Z}_p^* . The *r*-th derivative is taken in direction $\langle \chi \rangle$:

$$L_p^{(r)}(\Phi,\chi^s) = \int_{\mathbb{Z}_p^*} \chi^s(z) (\log z)^r d\mu(z).$$

In the argument list,

• mu is as returned by mspadicmoments (distributions attached to Φ by restriction to discs $a + p^{\nu} \mathbb{Z}_p$, (a, p) = 1).

• $s = [s_1, s_2]$ with $s_1 \in \mathbb{Z} \subset \mathbb{Z}_p$ and $s_2 modp - 1$ or $s_2 mod2$ for p = 2, encoding the *p*-adic character $\chi^s := \langle \chi \rangle^{s_1} \tau^{s_2}$; here χ is the cyclotomic character from $Gal(\mathbb{Q}_p(\mu_{p^oo})/\mathbb{Q}_p)$ to \mathbb{Z}_p^* , and τ is the Teichmüller character (for p > 2 and the character of order 2 on $(\mathbb{Z}/4\mathbb{Z})^*$ if p = 2); for convenience, the character [s, s] can also be represented by the integer s.

When a_p is a *p*-adic unit, L_p takes its values in \mathbb{Q}_p . When a_p is not a unit, it takes its values in the two-dimensional \mathbb{Q}_p -vector space $D_{cris}(M(\phi))$ where $M(\phi)$ is the "motive" attached to ϕ , and we return the two *p*-adic components with respect to some fixed \mathbb{Q}_p -basis.

```
? M = msinit(3,6,1); phi=[5, -3, -1]~;
? msissymbol(M,phi)
\%2 = 1
? Mp = mspadicinit(M, 5, 4);
? mu = mspadicmoments(Mp, phi); \\ no twist
\\ End of initializations
? mspadicL(mu,0) \\ L_p(chi^0)
\%5 = 5 + 2*5^2 + 2*5^3 + 2*5^4 + \dots
? mspadicL(mu,1) \\ L_p(chi), zero for parity reasons
%6 = [0(5^13)]~
? mspadicL(mu,2) \\ L_p(chi^2)
\%7 = 3 + 4*5 + 4*5^2 + 3*5^5 + \ldots
? mspadicL(mu,[0,2]) \\ L_p(tau^2)
\%8 = 3 + 5 + 2*5^2 + 2*5^3 + \dots
? mspadicL(mu, [1,0]) \\ L_p(<chi>)
\%9 = 3*5 + 2*5^2 + 5^3 + 2*5^7 + 5^8 + 5^{10} + 2*5^{11} + 0(5^{13})
? mspadicL(mu,0,1) \\ L_p'(chi^0)
\%10 = 2*5 + 4*5^2 + 3*5^3 + \ldots
? mspadicL(mu, 2, 1) \\ L_p'(chi^2)
\%11 = 4*5 + 3*5^2 + 5^3 + 5^4 + \dots
```

Now several quadratic twists: mstooms is indicated.

```
? PHI = mstooms(Mp,phi);
? mu = mspadicmoments(Mp, PHI, 12); \\ twist by 12
? mspadicL(mu)
%14 = 5 + 5^2 + 5^3 + 2*5^4 + ...
? mu = mspadicmoments(Mp, PHI, 8); \\ twist by 8
? mspadicL(mu)
%16 = 2 + 3*5 + 3*5^2 + 2*5^4 + ...
? mu = mspadicmoments(Mp, PHI, -3); \\ twist by -3 < 0
? mspadicL(mu)
%18 = 0(5^13) \\ always 0, phi is in the + part and D < 0</pre>
```

One can locate interesting symbols of level N and weight k with msnew and mssplit. Note that instead of a symbol, one can input a 1-dimensional Hecke-subspace from mssplit: the function will automatically use the underlying basis vector.

```
? M=msinit(5,4,1); \\ M_4(Gamma_0(5))^+
? L = mssplit(M, msnew(M)); \\ list of irreducible Hecke-subspaces
? phi = L[1]; \\ one Galois orbit of newforms
? #phi[1] \\... this one is rational
%4 = 1
? Mp = mspadicinit(M, 3, 4);
```

mspadicinit(M, p, n, flag)

M being a full modular symbol space, as given by msinit, and *p* a prime, initialize technical data needed to compute with overconvergent modular symbols, modulo p^n . If flag is unset, allow all symbols; else initialize only for a restricted range of symbols depending on flag: if flag = 0 restrict to ordinary symbols, else restrict to symbols ϕ such that $T_p(\phi) = a_p \phi$, with $v_p(a_p) \ge flag$, which is faster as flag increases. (The fastest initialization is obtained for flag = 0 where we only allow ordinary symbols.) For supersingular eigensymbols, such that $p ||a_p$, we must further assume that *p* does not divide the level.

? E = ellinit("11a1"); ? [M,phi] = msfromell(E,1); ? ellap(E,3) %3 = -1 ? Mp = mspadicinit(M, 3, 10, 0); \\ commit to ordinary symbols ? PHI = mstooms(Mp,phi);

If we restrict the range of allowed symbols with *flag* (for faster initialization), exceptions will occur if $v_p(a_p)$ violates this bound:

```
? E = ellinit("15a1");
? [M,phi] = msfromell(E,1);
? ellap(E,7)
%3 = 0
? Mp = mspadicinit(M,7,5,0); \\ restrict to ordinary symbols
? PHI = mstooms(Mp,phi)
*** at top-level: PHI=mstooms(Mp,phi)
*** ^_____
*** mstooms: incorrect type in mstooms [v_p(ap) > mspadicinit flag] (t_VEC).
? Mp = mspadicinit(M,7,5); \\ no restriction
? PHI = mstooms(Mp,phi);
```

This function uses $O(N^2(n+k)^2p)$ memory, where N is the level of M.

mspadicmoments(Mp, PHI, D)

Given Mp from mspadicinit, an overconvergent eigensymbol PHI from mstooms and a fundamental discriminant D coprime to p, let PHI^D denote the twisted symbol. This function computes the distribution $\mu = PHI^D([0] - oo]) ||\mathbb{Z}_p^*$ restricted to \mathbb{Z}_p^* . More precisely, it returns the moments of the p-1 distributions $PHI^D([0] - [oo]) ||(a + p\mathbb{Z}_p), 0 < a < p$. We also allow PHI to be given as a classical symbol, which is then lifted to an overconvergent symbol by mstooms; but this is wasteful if more than one twist is later needed. The returned data μ (*p*-adic distributions attached to PHI) can then be used in mspadicL or mspadicseries. This precomputation allows to quickly compute derivatives of different orders or values at different characters.

```
? M = msinit(3,6, 1);
? phi = [5,-3,-1]~;
? msissymbol(M, phi)
%3 = 1
? p = 5; mshecke(M,p) * phi \\ eigenvector of T_5, a_5 = 6
%4 = [30, -18, -6]~
? Mp = mspadicinit(M, p, 10, 0); \\ restrict to ordinary symbols, mod p^10
? PHI = mstooms(Mp, phi);
? mu = mspadicmoments(Mp, PHI);
? mspadicL(mu)
%8 = 5 + 2*5^2 + 2*5^3 + ...
? mu = mspadicmoments(Mp, PHI, 12); \\ twist by 12
? mspadicL(mu)
%10 = 5 + 5^2 + 5^3 + 2*5^4 + ...
```

mspadicseries(mu, i)

Let Φ be the *p*-adic distribution-valued overconvergent symbol attached to a modular symbol ϕ for $\Gamma_0(N)$ (eigenvector for $T_N(p)$ for the eigenvalue a_p). If μ is the distribution on \mathbb{Z}_p^* defined by the restriction of $\Phi([oo] - [0])$ to \mathbb{Z}_p^* , let

$${}_{p}^{L}(\mu,\tau^{i})(x) = \int_{\mathbb{Z}_{p}^{*}} \tau^{i}(t)(1+x)^{\log_{p}(t)/\log_{p}(u)} d\mu(t)$$

Here, τ is the Teichmüller character and u is a specific multiplicative generator of $1 + 2p\mathbb{Z}_p$. (Namely 1 + p if p > 2 or 5 if p = 2.) To explain the formula, let $G_o o := Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$, let $\chi : G_o o \to \mathbb{Z}_p^*$ be the cyclotomic character (isomorphism) and γ the element of $G_o o$ such that $\chi(\gamma) = u$; then $\chi(\gamma)^{\log_p(t)/\log_p(u)} = \langle t \rangle$.

The *p*-padic precision of individual terms is maximal given the precision of the overconvergent symbol μ .

```
? [M,phi] = msfromell(ellinit("17a1"),1);
? Mp = mspadicinit(M, 5,7);
? mu = mspadicmoments(Mp, phi,1); \\ overconvergent symbol
? mspadicseries(mu)
%4 = (4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 4*5^6 + 3*5^7 + 0(5^9)) \
+ (3 + 3*5 + 5^2 + 5^3 + 2*5^4 + 5^6 + 0(5^7))*x \
+ (2 + 3*5 + 5^2 + 4*5^3 + 2*5^4 + 0(5^5))*x^2 \
+ (3 + 4*5 + 4*5^2 + 0(5^3))*x^3 \
+ (3 + 0(5))*x^4 + 0(x^5)
```

An example with nonzero Teichmüller:

```
? [M,phi] = msfromell(ellinit("11a1"),1);
? Mp = mspadicinit(M, 3,10);
? mu = mspadicmoments(Mp, phi,1);
? mspadicseries(mu, 2)
%4 = (2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + 3^7 + 3^10 + 3^11 + 0(3^12)) \
+ (1 + 3 + 2*3^2 + 3^3 + 3^5 + 2*3^6 + 2*3^8 + 0(3^9))*x \
+ (1 + 2*3 + 3^4 + 2*3^5 + 0(3^6))*x^2 \
+ (3 + 0(3^2))*x^3 + 0(x^4)
```

Supersingular example (not checked)

```
? E = ellinit("17a1"); ellap(E,3)
%1 = 0
? [M,phi] = msfromell(E,1);
? Mp = mspadicinit(M, 3,7);
? mu = mspadicmoments(Mp, phi,1);
? mspadicseries(mu)
%5 = [(2*3^-1 + 1 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 0(3^7)) \
+ (2 + 3^3 + 0(3^5))*x \
+ (1 + 2*3 + 0(3^2))*x^2 + 0(x^3), \
(3^-1 + 1 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 0(3^7)) \
+ (1 + 2*3 + 2*3^2 + 3^3 + 2*3^4 + 0(3^5))*x \
+ (3^-2 + 3^-1 + 0(3^2))*x^2 + 0(3^-2)*x^3 + 0(x^4)]
```

Example with a twist:

```
? E = ellinit("11a1");
? [M,phi] = msfromell(E,1);
? Mp = mspadicinit(M, 3,10);
? mu = mspadicmoments(Mp, phi,5); \\ twist by 5
? L = mspadicseries(mu)
%5 = (2*3^2 + 2*3^4 + 3^5 + 3^6 + 2*3^7 + 2*3^10 + 0(3^12)) \
+ (2*3^2 + 2*3^6 + 3^7 + 3^8 + 0(3^9))*x \
+ (3^3 + 0(3^6))*x^2 + 0(3^2)*x^3 + 0(x^4)
? mspadicL(mu)
%6 = [2*3^2 + 2*3^4 + 3^5 + 3^6 + 2*3^7 + 2*3^10 + 0(3^12)]~
? ellpadicL(E,3,10,,5)
%7 = 2 + 2*3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^6 + 2*3^7 + 0(3^10)
? mspadicseries(mu,1) \\ must be 0
%8 = 0(3^12) + 0(3^9)*x + 0(3^6)*x^2 + 0(3^2)*x^3 + 0(x^4)
```

mspathgens(M)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by msinit, return a set of $\mathbb{Z}[G]$ generators for Δ_0 . The output is [g, R], where g is a minimal system of generators and R the vector of $\mathbb{Z}[G]$ relations between the given generators. A relation is coded by a vector of pairs $[a_i, i]$ with $a_i \in \mathbb{Z}[G]$ and i the
index of a generator, so that $\sum_i a_i g[i] = 0$.

An element [v] - [u] in Δ_0 is coded by the "path" [u, v], where oo denotes the point at infinity (1 : 0) on the projective line. An element of $\mathbb{Z}[G]$ is either an integer $n (= n[id_2])$ or a "factorization matrix": the first column contains distinct elements g_i of G and the second integers n_i and the matrix codes $\sum n_i[g_i]$:

```
? M = msinit(11,8); \\ M_8(Gamma_0(11))
? [g,R] = mspathgens(M);
? g
%3 = [[+oo, 0], [0, 1/3], [1/3, 1/2]] \\ 3 paths
? #R \\ a single relation
%4 = 1
? r = R[1]; #r \\ ...involving all 3 generators
%5 = 3
? r[1]
%6 = [[1, 1; [1, 1; 0, 1], -1], 1]
? r[2]
%7 = [[1, 1; [7, -2; 11, -3], -1], 2]
? r[3]
%8 = [[1, 1; [8, -3; 11, -4], -1], 3]
```

The given relation is of the form $\sum_i (1 - \gamma_i)g_i = 0$, with $\gamma_i \in \Gamma_0(11)$. There will always be a single relation involving all generators (corresponding to a round trip along all cusps), then relations involving a single generator (corresponding to 2 and 3-torsion elements in the group:

? M = msinit(2,8); \\ M_8(Gamma_0(2)) ? [g,R] = mspathgens(M); ? g %3 = [[+oo, 0], [0, 1]]

Note that the output depends only on the group G, not on the representation V.

mspathlog(M, p)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by msinit, encoding fixed $\mathbb{Z}[G]$ generators (g_i) of Δ_0 (see mspathgens). A path p = [a, b] between two elements in $\mathbb{P}^1(\mathbb{Q})$ corresponds to $[b] - [a] \in \Delta_0$. The path extremities a and b may be given as t_INT, t_FRAC or oo = (1:0). Finally, we also allow to input a path as a $2x^2$ integer matrix, whose first and second column give a and b respectively, with the convention [x, y] = (x: y) in $\mathbb{P}^1(\mathbb{Q})$.

Returns (p_i) in $\mathbb{Z}[G]$ such that $p = \sum_i p_i g_i$.

```
? M = msinit(2,8); \\ M_8(Gamma_0(2))
? [g,R] = mspathgens(M);
? g
%3 = [[+oo, 0], [0, 1]]
? p = mspathlog(M, [1/2,2/3]);
? p[1]
%5 =
[[1, 0; 2, 1] 1]
? p[2]
%6 =
[[1, 0; 0, 1] 1]
[[3, -1; 4, -1] 1]
? mspathlog(M, [1,2;2,3]) == p \\ give path via a 2x2 matrix
%7 = 1
```

Note that the output depends only on the group G, not on the representation V.

mspetersson(M, F, G)

M being a full modular symbol space for $\Gamma = \Gamma_0(N)$, as given by msinit, calculate the intersection product F, G of modular symbols F and G on $M = \text{Hom}_{\Gamma}(\Delta_0, V_k)$ extended to an hermitian bilinear form on $M \otimes \mathbb{C}$ whose radical is the Eisenstein subspace of M.

Suppose that f_1 and f_2 are two parabolic forms. Let F_1 and F_2 be the attached modular symbols

$$F_i(\delta) = \int_{\delta} f_i(z) . (zX + Y)^{k-2} dz$$

and let $F_1^{\mathbb{R}}$, $F_2^{\mathbb{R}}$ be the attached real modular symbols

$$F_i^{\mathbb{R}}(\delta) = \int_{\delta} \Re(f_i(z).(zX+Y)^{k-2}dz)$$

Then we have

$$= -2 (2i)^{k-2} \cdot \Im (< f_1, f_2 >_{Petersson})$$

and

$$F_1, \bar{F}_2 = (2i)^{k-2} < f_1, f_2 >_{Petersson}$$

In weight 2, the intersection product F, G has integer values on the \mathbb{Z} -structure on M given by mslattice and defines a Riemann form on $H^1_{par}(\Gamma, \mathbb{R})$.

For user convenience, we allow F and G to be matrices and return the attached Gram matrix. If F is omitted: treat it as the full modular space attached to M; if G is omitted, take it equal to F.

```
? M = msinit(37,2);
? C = mscuspidal(M)[1];
? mspetersson(M, C)
%3 =
[ 0 -17 -8 -17]
[17 0 -8 -25]
[ 8 8 0 -17]
[17 25 17 0]
? mspetersson(M, mslattice(M,C))
%4 =
[0 -1 \ 0 -1]
[1 \ 0 \ 0 \ -1]
[0 \ 0 \ 0 \ -1]
[1 \ 1 \ 1 \ 0]
? E = ellinit("33a1");
? [M,xpm] = msfromell(E); [xp,xm,L] = xpm;
? mspetersson(M, mslattice(M,L))
%7 =
[0 -3]
[3 0]
? ellmoddegree(E)
\%8 = [3, -126]
```

The coefficient 3 in the matrix is the degree of the modular parametrization.

mspolygon(M, flag)

M describes a subgroup G of finite index in the modular group $PSL_2(\mathbb{Z})$, as given by msinit or a positive integer N (encoding the group $G = \Gamma_0(N)$), or by msfarey (arbitrary subgroup). Return an hyperbolic polygon (Farey symbol) attached to G. More precisely:

- Its vertices are an ordered list in $\mathbb{P}^1(\mathbb{Q})$ and contain a representatives of all cusps.
- Its edges are hyperbolic arcs joining two consecutive vertices; each edge e is labelled by an integer $\mu(e) \in oo, 2, 3$.
- Given a path (a, b) between two elements of P¹(Q), let (a, b) = (b, a) be the opposite path. There is an involution e → e^{*} on the edges. We have μ(e) = oo if and only if e! = e^{*}; when μ(e)! = 3, e is G-equivalent to e^{*}, i.e. there exists γ_e ∈ G such that e = γ_ee^{*}; if μ(e) = 3 there exists γ_e ∈ G of order 3 such that the hyperbolic triangle (e, γ_ee, γ_e²e) is invariant by γ_e. In all cases, to each edge we have attached γ_e ∈ G of order μ(e).

The polygon is given by a triple [E, A, g]

- The list E of its consecutive edges as matrices in $M_2(\mathbb{Z})$.
- The permutation A attached to the involution: if e = E[i] is the *i*-th edge, then A[i] is the index of e^* in E.

• The list g of pairing matrices γ_e . Remark that $\gamma_{e^*} = \gamma_e^{-1}$ if $\mu(e)! = 3$, i.e., $g[i]^{-1} = g[A[i]]$ whenever $i! = A[i] (\mu(g[i]) = 1)$ or $\mu(g[i]) = 2 (g[i]^2 = 1)$. Modulo these trivial relations, the pairing matrices form a system of independent generators of G. Note that γ_e is elliptic if and only if $e^* = e$.

The above data yields a fundamental domain for G acting on Poincaré's half-plane: take the convex hull of the polygon defined by

- The edges in E such that $e! = e^*$ or $e^* = e$, where the pairing matrix γ_e has order 2;
- The edges (r, t) and (t, s) where the edge $e = (r, s) \in E$ is such that $e = e^*$ and γ_e has order 3 and the triangle (r, t, s) is the image of $(0, \exp(2i\pi/3), oo)$ by some element of $PSL_2(\mathbb{Q})$ formed around the edge.

Binary digits of flag mean:

1: return a normalized hyperbolic polygon if set, else a polygon with unimodular edges (matrices of determinant 1). A polygon is normalized in the sense of compact orientable surfaces if the distance $d(a, a^*)$ between an edge a and its image by the involution a^* is less than 2, with equality if and only if a is *linked* with another edge b (a, b, a^* et b^* appear consecutively in E up to cyclic permutation). In particular, the vertices of all edges such that that $d(a, a^*)! = 1$ (distance is 0 or 2) are all equivalent to 0 modulo G. The external vertices of aa^* such that $d(a, a^*) = 1$ are also equivalent to 0; the internal vertices $a \cap a^*$ (a single point), together with 0, form a system of representatives of the cusps of G

 $mathbbP^1(\mathbb{Q})$. This is useful to compute the homology group $H_1(G,\mathbb{Z})$ as it gives a symplectic basis for the intersection pairing. In this case, the number of parabolic matrices (trace 2) in the system of generators G is 2(t-1), where t is the number of non equivalent cusps for G. This is currently only implemented for $G = \Gamma_0(N)$.

2: add graphical representations (in LaTeX form) for the hyperbolic polygon in Poincaré's half-space and the involution $a \rightarrow a^*$ of the Farey symbol. The corresponding character strings can be included in a LaTeX document provided the preamble contains \usepackage{ tikz}.

```
? [V,A,g] = mspolygon(3);
? V
\%2 = [[-1, 1; -1, 0], [1, 0; 0, 1], [0, 1; -1, 1]]
? A
%3 = Vecsmall([2, 1, 3])
? g
\%4 = [[-1, -1; 0, -1], [1, -1; 0, 1], [1, -1; 3, -2]]
? [V,A,g, D1,D2] = mspolygon(11,2); \ D1 and D2 contains pictures
? {write("F.tex",
 "\\documentclass{article}\\usepackage{tikz}\\begin{document}"
D1, "\n", D2,
 "\\end{document}");}
? [V1,A1] = mspolygon(6,1); \\ normalized
? V1
\%8 = [[-1, 1; -1, 0], [1, 0; 0, 1], [0, 1; -1, 3],
[1, -2; 3, -5], [-2, 1; -5, 2], [1, -1; 2, -1]]
? A1
\%9 = \text{Vecsmall}([2, 1, 4, 3, 6, 5])
? [V0,A0] = mspolygon(6); \ not normalized V[3]^* = V[6], d(V[3],V[6]) = 3
? A0
%11 = Vecsmall([2, 1, 6, 5, 4, 3])
? [V,A] = mspolygon(14, 1);
? A
%13 = Vecsmall([2, 1, 4, 3, 6, 5, 9, 10, 7, 8])
```

One can see from this last example that the (normalized) polygon has the form

$$(a_1, a_1^*, a_2, a_2^*, a_3, a_3^*, a_4, a_5, a_4^*, a_5^*),$$

that $X_0(14)$ is of genus 1 (in general the genus is the number of blocks of the form aba^*b^*), has no elliptic points (A has no fixed point) and 4 cusps (number of blocks of the form aa^* plus 1). The vertices of edges a_4 and a_5 all project to 0 in $X_0(14)$: the paths a_4 and a_5 project as loops in $X_0(14)$ and give a symplectic basis of the homology $H_1(X_0(14), \mathbb{Z})$.

? [V,A] = mspolygon(15); ? apply(matdet, V) \\ all unimodular %2 = [1, 1, 1, 1, 1, 1, 1, 1, 1] ? [V,A] = mspolygon(15,1); ? apply(matdet, V) \\ normalized polygon but no longer unimodular edges %4 = [1, 1, 1, 1, 2, 2, 47, 11, 47, 11]

msqexpansion(M, projH, serprec)

M being a full modular symbol space, as given by msinit, and *projH* being a projector on a Hecke-simple subspace (as given by mssplit), return the Fourier coefficients a_n , $n \le B$ of the corresponding normalized newform. If B is omitted, use seriesprecision.

This function uses a naive $O(B^2d^3)$ algorithm, where d = O(kN) is the dimension of $M_k(\Gamma_0(N))$.

? M = msinit(11,2, 1); \\ M_2(Gamma_0(11))^+ ? L = mssplit(M, msnew(M)); ? msqexpansion(M,L[1], 20) %3 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2] ? ellan(ellinit("11a1"), 20) %4 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]

The shortcut msqexpansion(M, s, B) is available for a symbol s, provided it is a Hecke eigenvector:

? E = ellinit("11a1"); ? [M,S] = msfromell(E); [sp,sm] = S; ? msqexpansion(M,sp,10) \\ in the + eigenspace %3 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2] ? msqexpansion(M,sm,10) \\ in the - eigenspace %4 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2] ? ellan(E, 10) %5 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]

mssplit(M, H, dimlim)

Let M denote a full modular symbol space, as given by msinit(N, k, 1) or msinit(N, k, -1) and let H be a Hecke-stable subspace of msnew(M) (the full new subspace if H is omitted). This function splits H into Hecke-simple subspaces. If dimlim is present and positive, restrict to subspaces of dimension $\leq = dimlim$. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

```
? M = msinit(11,8, 1); \\ M_8(Gamma_0(11))^+
? L = mssplit(M); \\ split msnew(M)
? #L
%3 = 2
? f = msqexpansion(M,L[1],5); f[1].mod
%4 = x^2 + 8*x - 44
```

```
? lift(f)
%5 = [1, x, -6*x - 27, -8*x - 84, 20*x - 155]
? g = msqexpansion(M,L[2],5); g[1].mod
%6 = x^4 - 558*x^2 + 140*x + 51744
```

To a Hecke-simple subspace corresponds an orbit of (normalized) newforms, defined over a number field. In the above example, we printed the polynomials defining the said fields, as well as the first 5 Fourier coefficients (at the infinite cusp) of one such form.

msstar(M, H)

M being a full modular symbol space, as given by msinit, return the matrix of the * involution, induced by complex conjugation, acting on the (stable) subspace H (M if omitted).

```
? M = msinit(11,2); \\ M_2(Gamma_0(11))
? w = msstar(M);
? w^2 == 1
%3 = 1
```

mstooms(Mp, phi)

Given Mp from mspadicinit, lift the (classical) eigen symbol phi to a *p*-adic distribution-valued overconvergent symbol in the sense of Pollack and Stevens. More precisely, let ϕ belong to the space W of modular symbols of level N, $v_p(N) \leq 1$, and weight k which is an eigenvector for the Hecke operator $T_N(p)$ for a nonzero eigenvalue a_p and let $N_0 = lcm(N, p)$.

Under the action of $T_{N_0}(p)$, ϕ generates a subspace W_{ϕ} of dimension 1 (if p || N) or 2 (if p does not divide N) in the space of modular symbols of level N_0 .

Let $V_p = [p, 0; 0, 1]$ and $C_p = [a_p, p^{k-1}; -1, 0]$. When p does not divide N and a_p is divisible by p, mstooms returns the lift Φ of $(\phi, \phi ||_k V_p)$ such that

$$T_{N_0}(p)\Phi = C_p\Phi$$

When p does not divide N and a_p is not divisible by p, mstooms returns the lift Φ of $\phi - \alpha^{-1} \phi ||_k V_p$ which is an eigenvector of $T_{N_0}(p)$ for the unit eigenvalue where $\alpha^2 - a_p \alpha + p^{k-1} = 0$.

The resulting overconvergent eigensymbol can then be used in mspadicmoments, then mspadicL or mspadicseries.

```
? M = msinit(3,6, 1); p = 5;
? Tp = mshecke(M, p); factor(charpoly(Tp))
%2 =
[x - 3126 2]
[ x - 6 1]
? phi = matker(Tp - 6)[,1] \\ generator of p-Eigenspace, a_p = 6
%3 = [5, -3, -1]~
? Mp = mspadicinit(M, p, 10, 0); \\ restrict to ordinary symbols, mod p^10
? PHI = mstooms(Mp, phi);
? mu = mspadicmoments(Mp, PHI);
? mspadicL(mu)
%7 = 5 + 2*5^2 + 2*5^3 + ...
```

A non ordinary symbol.

```
? M = msinit(4.6.1): p = 3:
? Tp = mshecke(M, p); factor(charpoly(Tp))
%2 =
[x - 244 3]
[x + 12 1]
? phi = matker(Tp + 12)[,1] \setminus a_p = -12 is divisible by p = 3
\%3 = [-1/32, -1/4, -1/32, 1] \sim
? msissymbol(M,phi)
\%4 = 1
? Mp = mspadicinit(M,3,5,0);
? PHI = mstooms(Mp,phi);
 *** at top-level: PHI=mstooms(Mp,phi)
 *** mstooms: incorrect type in mstooms [v_p(ap) > mspadicinit flag] (t_VEC).
? Mp = mspadicinit(M,3,5,1);
? PHI = mstooms(Mp,phi);
```

newtonpoly(*x*, *p*)

Gives the vector of the slopes of the Newton polygon of the polynomial x with respect to the prime number p. The n components of the vector are in decreasing order, where n is equal to the degree of x. Vertical slopes occur iff the constant coefficient of x is zero and are denoted by +00.

nextprime(x)

Finds the smallest pseudoprime (see ispseudoprime) greater than or equal to x. x can be of any real type. Note that if x is a pseudoprime, this function returns x and not the smallest pseudoprime strictly larger than x. To rigorously prove that the result is prime, use isprime.

nfalgtobasis(nf, x)

Given an algebraic number x in the number field nf, transforms it to a column vector on the integral basis :emphasis:`nf.zk`.

? nf = nfinit(y² + 4); ? nf.zk %2 = [1, 1/2*y] ? nfalgtobasis(nf, [1,1]~) %3 = [1, 1]~ ? nfalgtobasis(nf, y) %4 = [0, 2]~ ? nfalgtobasis(nf, Mod(y, y²+4)) %5 = [0, 2]~

This is the inverse function of nfbasistoalg.

nfbasis(T, dK)

Let T(X) be an irreducible polynomial with integral coefficients. This function returns an integral basis of the number field defined by T, that is a \mathbb{Z} -basis of its maximal order. If present, dK is set to the discriminant of the returned order. The basis elements are given as elements in $K = \mathbb{Q}[X]/(T)$, in Hermite normal form with respect to the \mathbb{Q} -basis $(1, X, ..., X^{\deg T-1})$ of K, lifted to $\mathbb{Q}[X]$. In particular its first element is always 1 and its *i*-th element is a polynomial of degree i - 1 whose leading coefficient is the inverse of an integer: the product of those integers is the index of $\mathbb{Z}[X]/(T)$ in the maximal order \mathbb{Z}_K :

```
? nfbasis(x^2 + 4) \setminus Z[X]/(T) has index 2 in Z_K %1 = [1, x/2]
```

```
? nfbasis(x^2 + 4, &D)
%2 = [1, x/2]
? D
%3 = -4
```

This function uses a modified version of the round 4 algorithm, due to David Ford, Sebastian Pauli and Xavier Roblot.

Local basis, orders maximal at certain primes.

Obtaining the maximal order is hard: it requires factoring the discriminant D of T. Obtaining an order which is maximal at a finite explicit set of primes is easy, but it may then be a strict suborder of the maximal order. To specify that we are interested in a given set of places only, we can replace the argument T by an argument [T, listP], where listP encodes the primes we are interested in: it must be a factorization matrix, a vector of integers or a single integer.

- Vector: we assume that it contains distinct *prime* numbers.
- Matrix: we assume that it is a two-column matrix of a (partial) factorization of D; namely the first column contains distinct *primes* and the second one the valuation of D at each of these primes.
- Integer B: this is replaced by the vector of primes up to B. Note that the function will use at least O(B) time: a small value, about 10^5 , should be enough for most applications. Values larger than 2^{32} are not supported.

In all these cases, the primes may or may not divide the discriminant D of T. The function then returns a \mathbb{Z} -basis of an order whose index is not divisible by any of these prime numbers. The result may actually be a global integral basis, in particular if all the prime divisors of the *field* discriminant are included, but this is not guaranteed! Note that nfinit has built-in support for such a check:

```
? K = nfinit([T, listP]);
? nfcertify(K) \\ we computed an actual maximal order
%2 = [];
```

The first line initializes a number field structure incorporating nfbasis([T, listP] in place of a proven integral basis. The second line certifies that the resulting structure is correct. This allows to create an nf structure attached to the number field $K = \mathbb{Q}[X]/(T)$, when the discriminant of T cannot be factored completely, whereas the prime divisors of discK are known. If present, the argument dK is set to the discriminant of the returned order, and is equal to the field discriminant if and only if the order is maximal.

Of course, if *listP* contains a single prime number p, the function returns a local integral basis for $\mathbb{Z}_p[X]/(T)$:

? nfbasis(x^2+x-1001)
%1 = [1, 1/3*x - 1/3]
? nfbasis([x^2+x-1001, [2]])
%2 = [1, x]

The following function computes the index i_T of $\mathbb{Z}[X]/(T)$ in the order generated by the \mathbb{Z} -basis B:

nfbasisindex(T, B) = vecprod([denominator(pollead(Q)) | Q <- B]);</pre>

In particular, B is a basis of the maximal order if and only if $poldisc(T)/i_T^2$ is equal to the field discriminant. More generally, this formula gives the square of index of the order given by B in \mathbb{Z}_K . For instance, assume that P is a vector of prime numbers containing (at least) all prime divisors of the field discriminant, then the following construct allows to provably compute the field discriminant and to check whether the returned basis is actually a basis of the maximal order

```
? B = nfbasis([T, P], &D);
? dK = sign(D) * vecprod([p^valuation(D,p) | p<-P]);
? dK * nfbasisindex(T, B)^2 == poldisc(T)
```

The variable dK contains the field discriminant and the last command returns 1 if and only if B is a \mathbb{Z} -basis of the maximal order. Of course, the nfinit / nfcertify approach is simpler, but it is also more costly.

The Buchmann-Lenstra algorithm.

We now complicate the picture: it is in fact allowed to include *composite* numbers instead of primes in listP (Vector or Matrix case), provided they are pairwise coprime. The result may still be a correct integral basis if the field discriminant factors completely over the actual primes in the list; again, this is not guaranteed. Adding a composite C such that C^2 divides D may help because when we consider C as a prime and run the algorithm, two good things can happen: either we succeed in proving that no prime dividing C can divide the index (without actually needing to find those primes), or the computation exhibits a nontrivial zero divisor, thereby factoring C and we go on with the refined factorization. (Note that including a C such that C^2 does not divide D is useless.) If neither happen, then the computed basis need not generate the maximal order. Here is an example:

```
? B = 10^5;
? listP = factor(poldisc(T), B); \\ primes <= B dividing D + cofactor
? basis = nfbasis([T, listP], &D)
```

If the computed discriminant D factors completely over the primes less than B (together with the primes contained in the addprimes table), then everything is certified: D is the field discriminant and basis generates the maximal order. This can be tested as follows:

F = factor(D, B); P = F[,1]; E = F[,2]; for (i = 1, #P, if (P[i] > B && !isprime(P[i]), warning("nf may be incorrect")));

This is a sufficient but not a necessary condition, hence the warning, instead of an error.

The function nfcertify speeds up and automates the above process:

```
? B = 10^5;
? nf = nfinit([T, B]);
? nfcertify(nf)
%3 = [] \\ nf is unconditionally correct
? [basis, disc] = [nf.zk, nf.disc];
```

nfbasistoalg(*nf*, *x*)

Given an algebraic number x in the number field nf, transforms it into t_POLMOD form.

```
? nf = nfinit(y<sup>2</sup> + 4);
? nf.zk
%2 = [1, 1/2*y]
? nfbasistoalg(nf, [1,1]~)
%3 = Mod(1/2*y + 1, y<sup>2</sup> + 4)
? nfbasistoalg(nf, y)
%4 = Mod(y, y<sup>2</sup> + 4)
? nfbasistoalg(nf, Mod(y, y<sup>2</sup>+4))
%5 = Mod(y, y<sup>2</sup> + 4)
```

This is the inverse function of nfalgtobasis.

nfcertify(nf)

nf being as output by nfinit, checks whether the integer basis is known unconditionally. This is in particular useful when the argument to nfinit was of the form [T, listP], specifying a finite list of primes when p-maximality had to be proven, or a list of coprime integers to which Buchmann-Lenstra algorithm was to be applied.

The function returns a vector of coprime composite integers. If this vector is empty, then nf.zk and nf.disc are correct. Otherwise, the result is dubious. In order to obtain a certified result, one must completely factor each of the given integers, then addprime each of their prime factors, then check whether nfdisc(nf.pol) is equal to nf.disc.

nfcompositum(*nf*, *P*, *Q*, *flag*)

Let *nf* be a number field structure attached to the field *K* and let *P* and *Q* be squarefree polynomials in K[X] in the same variable. Outputs the simple factors of the étale *K*-algebra A = K[X,Y]/(P(X),Q(Y)). The factors are given by a list of polynomials *R* in K[X], attached to the number field K[X]/(R), and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. In this case, there will be a single factor R if and only if the number fields defined by P and Q are linearly disjoint (their intersection is K).

The binary digits of flag mean

1: outputs a vector of 4-component vectors [R, a, b, k], where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of K[X]/(R). Finally, k is a small integer such that b + ka = X modulo R.

2: assume that P and Q define number fields that are linearly disjoint: both polynomials are irreducible and the corresponding number fields have no common subfield besides K. This allows to save a costly factorization over K. In this case return the single simple factor instead of a vector with one element.

A compositum is often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is an example involving the field $K(\zeta_5, 5^{1/10}), K = \mathbb{Q}(\sqrt{5})$:

```
? K = nfinit(y^{2}-5);
? L = nfcompositum(K, x^5 - y, polcyclo(5), 1); \\ list of [R,a,b,k]
? [R, a] = L[1]; \setminus pick the single factor, extract R,a (ignore b,k)
? lift(R) \setminus defines the compositum
\%4 = x^{10} + (-5/2*y + 5/2)*x^9 + (-5*y + 20)*x^8 + (-20*y + 30)*x^7 + 
(-45/2*y + 145/2)*x^{6} + (-71/2*y + 121/2)*x^{5} + (-20*y + 60)*x^{4} + 
(-25*y + 5)*x^3 + 45*x^2 + (-5*y + 15)*x + (-2*y + 6)
? a^5 - y \setminus a fifth root of y
%5 = 0
? [T, X] = rnfpolredbest(K, R, 1);
? lift(T) \setminus simpler defining polynomial for K[x]/(R)
\%7 = x^{10} + (-11/2*y + 25/2)
? liftall(X) \setminus root of R in K[x]/(T(x))
\%8 = (3/4*y + 7/4)*x^{7} + (-1/2*y - 1)*x^{5} + 1/2*x^{2} + (1/4*y - 1/4)
? a = subst(a.pol, 'x, X); \setminus a in the new coordinates
? liftall(a)
\%10 = (-3/4*y - 7/4)*x^7 - 1/2*x^2
? a^5 - y
%11 = 0
```

The main variables of P and Q must be the same and have higher priority than that of nf (see varhigher and varlower).

nfdetint(*nf*, *x*)

Given a pseudo-matrix x, computes a nonzero ideal contained in (i.e. multiple of) the determinant of x. This is particularly useful in conjunction with nfhnfmod.

nfdisc(T)

field discriminant of the number field defined by the integral, preferably monic, irreducible polynomial T(X). Returns the discriminant of the number field $\mathbb{Q}[X]/(T)$, using the Round 4 algorithm.

Local discriminants, valuations at certain primes.

As in nfbasis, the argument T can be replaced by [T, listP], where listP is as in nfbasis: a vector of pairwise coprime integers (usually distinct primes), a factorization matrix, or a single integer. In that case, the function returns the discriminant of an order whose basis is given by nfbasis(T,listP), which need not be the maximal order, and whose valuation at a prime entry in listP is the same as the valuation of the field discriminant.

In particular, if listP is [p] for a prime p, we can return the p-adic discriminant of the maximal order of $\mathbb{Z}_p[X]/(T)$, as a power of p, as follows:

```
? padicdisc(T,p) = p^valuation(nfdisc([T,[p]]), p);
? nfdisc(x^2 + 6)
%2 = -24
? padicdisc(x^2 + 6, 2)
%3 = 8
? padicdisc(x^2 + 6, 3)
%4 = 3
```

The following function computes the discriminant of the maximal order under the assumption that P is a vector of prime numbers containing (at least) all prime divisors of the field discriminant:

```
globaldisc(T, P) =
{ my (D = nfdisc([T, P]));
  sign(D) * vecprod([p^valuation(D,p) | p <-P]);
}
? globaldisc(x^2 + 6, [2, 3, 5])
%1 = -24</pre>
```

nfdiscfactors(T)

Given a polynomial T with integer coefficients, return [D, faD] where D is nfdisc(T) and faD is the factorization of ||D||. All the variants [T,listP] are allowed (see ??nfdisc), in which case faD is the factorization of the discriminant underlying order (which need not be maximal at the primes not specified by listP) and the factorization may contain large composites.

```
? T = x^3 - 6021021*x^2 + 12072210077769*x - 8092423140177664432;
? [D,faD] = nfdiscfactors(T); print(faD); D
[3, 3; 500009, 2]
%2 = -6750243002187]
? T = x^3 + 9*x^2 + 27*x - 125014250689643346789780229390526092263790263725;
? [D,faD] = nfdiscfactors(T); print(faD); D
[3, 3; 1000003, 2]
%4 = -27000162000243
```

```
? [D,faD] = nfdiscfactors([T, 10^3]); print(faD)
[3, 3; 125007125141751093502187, 2]
```

In the final example, we only get a partial factorization, which is only guaranteed correct at primes $\langle = 10^3$.

The function also accept number field structures, for instance as output by nfinit, and returns the field discriminant and its factorization:

```
? T = x^3 + 9*x^2 + 27*x - 125014250689643346789780229390526092263790263725;
? nf = nfinit(T); [D,faD] = nfdiscfactors(T); print(faD); D
%2 = -27000162000243
? nf.disc
%3 = -27000162000243
```

nfeltadd(nf, x, y)

Given two elements x and y in nf, computes their sum x + y in the number field nf.

? nf = nfinit(1+x^2); ? nfeltadd(nf, 1, x) \\ 1 + I %2 = [1, 1]~

nfeltdiv(*nf*, *x*, *y*)

Given two elements x and y in nf, computes their quotient x/y in the number field nf.

nfeltdiveuc(*nf*, *x*, *y*)

Given two elements x and y in nf, computes an algebraic integer q in the number field nf such that the components of x - qy are reasonably small. In fact, this is functionally identical to round(nfdiv(:emphasis:`nf,x,y))`.

nfeltdivmodpr(nf, x, y, pr)

This function is obsolete, use nfmodpr.

Given two elements x and y in nf and pr a prime ideal in modpr format (see nfmodprinit), computes their quotient x/y modulo the prime ideal pr.

nfeltdivrem(*nf*, *x*, *y*)

Given two elements x and y in nf, gives a two-element row vector [q, r] such that x = qy + r, q is an algebraic integer in nf, and the components of r are reasonably small.

nfeltembed(*nf*, *x*, *pl*, *precision*)

Given an element x in the number field nf, return the (real or) complex embeddings of x specified by optional argument pl, at the current realprecision:

- *pl* omitted: return the vector of embeddings at all $r_1 + r_2$ places;
- *pl* an integer between 1 and r₁ + r₂: return the *i*-th embedding of x, attached to the *i*-th root of nf.pol, i.e. nf.roots:math:`[i]`;
- pl a vector or t_VECSMALL: return the vector of embeddings; the *i*-th entry gives the embedding at the place attached to the pl[i]-th real root of nf.pol.

```
? nf = nfinit('y^3 - 2);
? nf.sign
%2 = [1, 1]
? nfeltembed(nf, 'y)
%3 = [1.25992[...], -0.62996[...] + 1.09112[...]*I]]
? nfeltembed(nf, 'y, 1)
```

```
%4 = 1.25992[...]
? nfeltembed(nf, 'y, 3) \\ there are only 2 arch. places
 *** at top-level: nfeltembed(nf,'y,3)
 *** ^------
 *** nfeltembed: domain error in nfeltembed: index > 2
```

nfeltmod(nf, x, y)

Given two elements x and y in nf, computes an element r of nf of the form r = x - qy with q and algebraic integer, and such that r is small. This is functionally identical to

$$x - nfmul(nf, round(nfdiv(nf, x, y)), y).$$

nfeltmul(*nf*, *x*, *y*)

Given two elements x and y in nf, computes their product x * y in the number field nf.

nfeltmulmodpr(nf, x, y, pr)

This function is obsolete, use nfmodpr.

Given two elements x and y in nf and pr a prime ideal in modpr format (see nfmodprinit), computes their product x * y modulo the prime ideal pr.

nfeltnorm(nf, x)

Returns the absolute norm of x.

nfeltpow(*nf*, *x*, *k*)

Given an element x in nf, and a positive or negative integer k, computes x^k in the number field nf.

nfeltpowmodpr(*nf*, *x*, *k*, *pr*)

This function is obsolete, use nfmodpr.

Given an element x in nf, an integer k and a prime ideal pr in modpr format (see nfmodprinit), computes x^k modulo the prime ideal pr.

nfeltreduce(*nf*, *a*, *id*)

Given an ideal *id* in Hermite normal form and an element *a* of the number field nf, finds an element *r* in nf such that a - r belongs to the ideal and *r* is small.

nfeltreducemodpr(nf, x, pr)

This function is obsolete, use nfmodpr.

Given an element x of the number field nf and a prime ideal pr in modpr format compute a canonical representative for the class of x modulo pr.

nfeltsign(*nf*, *x*, *pl*)

Given an element x in the number field nf, returns the signs of the real embeddings of x specified by optional argument pl:

- *pl* omitted: return the vector of signs at all r_1 real places;
- *pl* an integer between 1 and *r*₁: return the sign of the *i*-th embedding of *x*, attached to the *i*-th real root of nf.pol, i.e. nf.roots:math:`[i]`;
- pl a vector or t_VECSMALL: return the vector of signs; the *i*-th entry gives the sign at the real place attached to the pl[i]-th real root of nf.pol.

```
? nf = nfinit(polsubcyclo(11,5,'y)); \\ Q(cos(2 pi/11))
? nf.sign
%2 = [5, 0]
? x = Mod('y, nf.pol);
? nfeltsign(nf, x)
%4 = [-1, -1, -1, 1, 1]
? nfeltsign(nf, x, 1)
%5 = -1
? nfeltsign(nf, x, [1..4])
%6 = [-1, -1, -1, 1]
? nfeltsign(nf, x, 6) \\ there are only 5 real embeddings
*** at top-level: nfeltsign(nf,x,6)
*** ^------
*** nfeltsign: domain error in nfeltsign: index > 5
```

nfelttrace(*nf*, *x*)

Returns the absolute trace of x.

nfeltval(*nf*, *x*, *pr*, *y*)

Given an element x in nf and a prime ideal pr in the format output by idealprimedec, computes the valuation v at pr of the element x. The valuation of 0 is +00.

? nf = nfinit(x^2 + 1); ? P = idealprimedec(nf, 2)[1]; ? nfeltval(nf, x+1, P) %3 = 1

This particular valuation can also be obtained using idealval(:emphasis:`nf,x,:emphasis:pr)`, since x is then converted to a principal ideal.

If the y argument is present, sets $y = x\tau^v$, where τ is a fixed "anti-uniformizer" for pr: its valuation at pr is -1; its valuation is 0 at other prime ideals dividing :emphasis:`pr.p` and nonnegative at all other primes. In other words y is the part of x coprime to pr. If x is an algebraic integer, so is y.

? nfeltval(nf, x+1, P, &y); y %4 = [0, 1]~

For instance if $x = \prod_i x_i^{e_i}$ is known to be coprime to pr, where the x_i are algebraic integers and $e_i \in \mathbb{Z}$ then, if $v_i = nfeltval(nf, x_i, pr, y_i)$, we still have $x = \prod_i y_i^{e_i}$, where the y_i are still algebraic integers but now all of them are coprime to pr. They can then be mapped to the residue field of pr more efficiently than if the product had been expanded beforehand: we can reduce mod pr after each ring operation.

nffactor(nf, T)

Factorization of the univariate polynomial (or rational function) T over the number field nf given by nfinit; T has coefficients in nf (i.e. either scalar, polmod, polynomial or column vector). The factors are sorted by increasing degree.

The main variable of nf must be of *lower* priority than that of T, see priority (in the PARI manual). However if the polynomial defining the number field occurs explicitly in the coefficients of T as modulus of a t_POLMOD or as a t_POL coefficient, its main variable must be *the same* as the main variable of T. For example,

```
? nf = nfinit(y<sup>2</sup> + 1);
? nffactor(nf, x<sup>2</sup> + y); \\ OK
? nffactor(nf, x<sup>2</sup> + Mod(y, y<sup>2</sup>+1)); \\ OK
? nffactor(nf, x<sup>2</sup> + Mod(z, z<sup>2</sup>+1)); \\ WRONG
```

It is possible to input a defining polynomial for *nf* instead, but this is in general less efficient since parts of an nf structure will then be computed internally. This is useful in two situations: when you do not need the nf elsewhere, or when you cannot initialize an nf due to integer factorization difficulties when attempting to compute the field discriminant and maximal order. In all cases, the function runs in polynomial time using Belabas's variant of van Hoeij's algorithm, which copes with hundreds of modular factors.

Caveat. nfinit([T, listP]) allows to compute in polynomial time a conditional *nf* structure, which sets nf. zk to an order which is not guaranteed to be maximal at all primes. Always either use nfcertify first (which may not run in polynomial time) or make sure to input nf.pol instead of the conditional *nf*: nffactor is able to recover in polynomial time in this case, instead of potentially missing a factor.

nffactorback(*nf*, *f*, *e*)

Gives back the nf element corresponding to a factorization. The integer 1 corresponds to the empty factorization.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization matrix.

```
? nf = nfinit(y<sup>2</sup>+1);
? nffactorback(nf, [3, y+1, [1,2]~], [1, 2, 3])
%2 = [12, -66]~
? 3 * (I+1)<sup>2</sup> * (1+2*I)<sup>3</sup>
%3 = 12 - 66*I
```

nffactormod(*nf*, *Q*, *pr*)

This routine is obsolete, use nfmodpr and factormod.

Factors the univariate polynomial Q modulo the prime ideal pr in the number field nf. The coefficients of Q belong to the number field (scalar, polynomial, even column vector) and the main variable of nf must be of lower priority than that of Q (see priority (in the PARI manual)). The prime ideal pr is either in idealprimedec or (preferred) modprimit format. The coefficients of the polynomial factors are lifted to elements of nf:

```
? K = nfinit(y<sup>2</sup>+1);
? P = idealprimedec(K, 3)[1];
? nffactormod(K, x<sup>2</sup> + y*x + 18*y+1, P)
%3 =
[x + (2*y + 1) 1]
[x + (2*y + 2) 1]
? P = nfmodprinit(K, P); \\ convert to nfmodprinit format
? nffactormod(K, x<sup>2</sup> + y*x + 18*y+1)
%5 =
[x + (2*y + 1) 1]
[x + (2*y + 2) 1]
```

Same result, of course, here about 10% faster due to the precomputation.

nfgaloisapply(*nf*, *aut*, *x*)

Let nf be a number field as output by nfinit, and let *aut* be a Galois automorphism of nf expressed by its image on the field generator (such automorphisms can be found using nfgaloisconj). The function computes the action of the automorphism *aut* on the object x in the number field; x can be a number field element, or an ideal (possibly extended). Because of possible confusion with elements and ideals, other vector or matrix arguments are forbidden.

```
? nf = nfinit(x^2+1);
? L = nfgaloisconj(nf)
%2 = [-x, x]~
? aut = L[1]; /* the nontrivial automorphism */
? nfgaloisapply(nf, aut, x)
%4 = Mod(-x, x^2 + 1)
? P = idealprimedec(nf,5); /* prime ideals above 5 */
? nfgaloisapply(nf, aut, P[2]) == P[1]
%6 = 0 \\ !!!!
? idealval(nf, nfgaloisapply(nf, aut, P[2]), P[1])
%7 = 1
```

The surprising failure of the equality test (%7) is due to the fact that although the corresponding prime ideals are equal, their representations are not. (A prime ideal is specified by a uniformizer, and there is no guarantee that applying automorphisms yields the same elements as a direct idealprimedec call.)

The automorphism can also be given as a column vector, representing the image of Mod(x, nf.pol) as an algebraic number. This last representation is more efficient and should be preferred if a given automorphism must be used in many such calls.

```
? nf = nfinit(x^3 - 37*x^2 + 74*x - 37);
? aut = nfgaloisconj(nf)[2]; \\ an automorphism in basistoalg form
%2 = -31/11*x^2 + 1109/11*x - 925/11
? AUT = nfalgtobasis(nf, aut); \\ same in algtobasis form
%3 = [16, -6, 5]~
? v = [1, 2, 3]~; nfgaloisapply(nf, aut, v) == nfgaloisapply(nf, AUT, v)
%4 = 1 \\ same result...
? for (i=1,10^5, nfgaloisapply(nf, aut, v))
time = 463 ms.
? for (i=1,10^5, nfgaloisapply(nf, AUT, v))
time = 343 ms. \\ but the latter is faster
```

nfgaloisconj(*nf*, *flag*, *d*, *precision*)

nf being a number field as output by nfinit, computes the conjugates of a root r of the nonconstant polynomial x = nf[1] expressed as polynomials in r. This also makes sense when the number field is not Galois since some conjugates may lie in the field. nf can simply be a polynomial.

If no flags or flag = 0, use a combination of flag 4 and 1 and the result is always complete. There is no point whatsoever in using the other flags.

If flag = 1, use nfroots: a little slow, but guaranteed to work in polynomial time.

If flag = 4, use galoisinit: very fast, but only applies to (most) Galois fields. If the field is Galois with weakly super-solvable Galois group (see galoisinit), return the complete list of automorphisms, else only the identity element. If present, d is assumed to be a multiple of the least common denominator of the conjugates expressed as polynomial in a root of *pol*.

This routine can only compute \mathbb{Q} -automorphisms, but it may be used to get K-automorphism for any base field K as follows:

```
rnfgaloisconj(nfK, R) = \\ K-automorphisms of L = K[X] / (R)
{
  my(polabs, N,al,S, ala,k, vR);
  R *= Mod(1, nfK.pol); \\ convert coeffs to polmod elts of K
  vR = variable(R);
```

```
al = Mod(variable(nfK.pol),nfK.pol);
[polabs,ala,k] = rnfequation(nfK, R, 1);
Rt = if(k==0,R,subst(R,vR,vR-al*k));
N = nfgaloisconj(polabs) % Rt; \\ Q-automorphisms of L
S = select(s->subst(Rt, vR, Mod(s,Rt)) == 0, N);
if (k==0, S, apply(s->subst(s,vR,vR+k*al)-k*al,S));
}
K = nfinit(y^2 + 7);
rnfgaloisconj(K, x^4 - y*x^3 - 3*x^2 + y*x + 1) \\ K-automorphisms of L
```

nfgrunwaldwang(*nf*, *Lpr*, *Ld*, *pl*, *v*)

Given *nf* a number field in *nf* or *bnf* format, a t_VEC *Lpr* of primes of *nf* and a t_VEC *Ld* of positive integers of the same length, a t_VECSMALL *pl* of length r_1 the number of real places of *nf*, computes a polynomial with coefficients in *nf* defining a cyclic extension of *nf* of minimal degree satisfying certain local conditions:

- at the prime Lpr[i], the extension has local degree a multiple of Ld[i];
- at the *i*-th real place of *nf*, it is complex if pl[i] = -1 (no condition if pl[i] = 0).

The extension has degree the LCM of the local degrees. Currently, the degree is restricted to be a prime power for the search, and to be prime for the construction because of the rnfkummer restrictions.

When nf is \mathbb{Q} , prime integers are accepted instead of prid structures. However, their primality is not checked and the behavior is undefined if you provide a composite number.

Warning. If the number field *nf* does not contain the *n*-th roots of unity where *n* is the degree of the extension to be computed, the function triggers the computation of the *bnf* of $nf(\zeta_n)$, which may be costly.

```
? nf = nfinit(y<sup>2-5</sup>);
? pr = idealprimedec(nf,13)[1];
? pol = nfgrunwaldwang(nf, [pr], [2], [0,-1], 'x)
%3 = x<sup>2</sup> + Mod(3/2*y + 13/2, y<sup>2</sup> - 5)
```

nfhilbert(*nf*, *a*, *b*, *pr*)

If pr is omitted, compute the global quadratic Hilbert symbol (a, b) in nf, that is 1 if $x^2 - ay^2 - bz^2$ has a non trivial solution (x, y, z) in nf, and -1 otherwise. Otherwise compute the local symbol modulo the prime ideal pr, as output by idealprimedec.

nfhnf(nf, x, flag)

Given a pseudo-matrix (A, I), finds a pseudo-basis (B, J) in Hermite normal form of the module it generates. If flag is nonzero, also return the transformation matrix U such that AU = [0||B].

nfhnfmod(*nf*, *x*, *detx*)

Given a pseudo-matrix (A, I) and an ideal *detx* which is contained in (read integral multiple of) the determinant of (A, I), finds a pseudo-basis in Hermite normal form of the module generated by (A, I). This avoids coefficient explosion. *detx* can be computed using the function nfdetint.

nfinit(pol, flag, precision)

pol being a nonconstant irreducible polynomial in $\mathbb{Q}[X]$, preferably monic and integral, initializes a number field structure (nf) attached to the field K defined by pol. As such, it's a technical object passed as the first argument to most nfxxx functions, but it contains some information which may be directly useful. Access to this information via member functions is preferred since the specific data organization given below may change in the future. Currently, nf is a row vector with 9 components:

nf[1] contains the polynomial *pol* (:emphasis:`nf.pol`).

nf[2] contains [r1, r2] (:emphasis:`nf.sign`, :emphasis:`nf.r1`, :emphasis:`nf.r2`), the number of real and complex places of K.

nf[3] contains the discriminant d(K) (:emphasis:`nf.disc`) of K.

nf[4] contains the index of nf[1] (:emphasis:`nf.index`), i.e. $[\mathbb{Z}_K : \mathbb{Z}[\theta]]$, where θ is any root of nf[1].

nf[5] is a vector containing 7 matrices M, G, roundG, T, MD, TI, MDI and a vector vP defined as follows:

* M is the (r1+r2)xn matrix whose columns represent the numerical values of the conjugates of the elements of the integral basis.

* G is an nxn matrix such that $T2 = {}^t GG$, where T2 is the quadratic form $T_2(x) = \sum ||\sigma(x)||^2$, σ running over the embeddings of K into \mathbb{C} .

* roundG is a rescaled copy of G, rounded to nearest integers.

* T is the nxn matrix whose coefficients are $Tr(\omega_i\omega_j)$ where the ω_i are the elements of the integral basis. Note also that $\det(T)$ is equal to the discriminant of the field K. Also, when understood as an ideal, the matrix T^{-1} generates the codifferent ideal.

* The columns of MD (:emphasis:`nf.diff`) express a \mathbb{Z} -basis of the different of K on the integral basis.

* TI is equal to the primitive part of T^{-1} , which has integral coefficients.

* *MDI* is a two-element representation (for faster ideal product) of d(K) times the codifferent ideal (:emphasis:`nf.disc:math:**nf*.codiff`, which is an integral ideal). This is used in idealinv.

* *vP* is the list of prime divisors of the field discriminant, i.e, the ramified primes (:emphasis:`nf.p`); nfdiscfactors(nf) is the preferred way to access that information.

nf[6] is the vector containing the r1 + r2 roots (:emphasis:`nf.roots`) of nf[1] corresponding to the r1 + r2 embeddings of the number field into \mathbb{C} (the first r1 components are real, the next r2 have positive imaginary part).

nf[7] is a \mathbb{Z} -basis for $d\mathbb{Z}_K$, where $d = [\mathbb{Z}_K : \mathbb{Z}(\theta)]$, expressed on the powers of θ . The multiplication by d ensures that all polynomials have integral coefficients and nf[7]/d (:emphasis:`nf.zk`) is an integral basis for \mathbb{Z}_K . Its first element is guaranteed to be 1. This basis is LLL-reduced with respect to T_2 (strictly speaking, it is a permutation of such a basis, due to the condition that the first element be 1).

nf[8] is the nxn integral matrix expressing the power basis in terms of the integral basis, and finally

nf[9] is the nxn^2 matrix giving the multiplication table of the integral basis.

If a non monic or non integral polynomial is input, nfinit will transform it, and return a structure attached to the new (monic integral) polynomial together with the attached change of variables, see flag = 3. It is allowed, though not very useful given the existence of nfnewprec, to input a *nf* or a *bnf* instead of a polynomial. It is also allowed to input a *rnf*, in which case an nf structure attached to the absolute defining polynomial polabs is returned (*flag* is then ignored).

```
? nf = nfinit(x^3 - 12); \\ initialize number field Q[X] / (X^3 - 12)
? nf.pol \\ defining polynomial
%2 = x^3 - 12
? nf.disc \\ field discriminant
%3 = -972
? nf.index \\ index of power basis order in maximal order
%4 = 2
? nf.zk \\ integer basis, lifted to Q[X]
%5 = [1, x, 1/2*x^2]
? nf.sign \\ signature
%6 = [1, 1]
? factor(abs(nf.disc )) \\ determines ramified primes
```

%7 =
[2 2]
[3 5]
? idealfactor(nf, 2)
%8 =
[[2, [0, 0, -1]~, 3, 1, [0, 1, 0]~] 3] \\ p_2^3

Huge discriminants, helping nfdisc.

In case *pol* has a huge discriminant which is difficult to factor, it is hard to compute from scratch the maximal order. The following special input formats are also accepted:

- [pol, B] where pol is a monic integral polynomial and B is the lift of an integer basis, as would be computed by nfbasis: a vector of polynomials with first element 1 (implicitly modulo pol). This is useful if the maximal order is known in advance.
- [pol, B, P] where pol and B are as above (a monic integral polynomial and the lift of an integer basis), and P is the list of ramified primes in the extension.
- [pol, listP] where pol is a rational polynomial and listP specifies a list of primes as in nfbasis. Instead of the maximal order, nfinit then computes an order which is maximal at these particular primes as well as the primes contained in the private prime table, see addprimes. The result has a good chance of being correct when the discriminant nf.disc factors completely over this set of primes but this is not guaranteed. The function nfcertify automates this:

```
? pol = polcompositum(x^5 - 101, polcyclo(7))[1];
? nf = nfinit( [pol, 10^3] );
? nfcertify(nf)
%3 = []
```

A priori, nf.zk defines an order which is only known to be maximal at all primes $\langle = 10^3$ (no prime $\langle = 10^3$ divides nf.index). The certification step proves the correctness of the computation. Had it failed, that particular nf structure could not have been trusted and may have caused routines using it to fail randomly. One particular function that remains trustworthy in all cases is idealprimedec when applied to a prime included in the above list of primes or, more generally, a prime not dividing any entry in nfcertify output.

If flag = 2: pol is changed into another polynomial P defining the same number field, which is as simple as can easily be found using the **polredbest** algorithm, and all the subsequent computations are done using this new polynomial. In particular, the first component of the result is the modified polynomial.

If flag = 3, apply polredbest as in case 2, but outputs [nf, Mod(a, P)], where nf is as before and Mod(a, P) = Mod(x, pol) gives the change of variables. This is implicit when pol is not monic or not integral: first a linear change of variables is performed, to get a monic integral polynomial, then polredbest.

nfisideal(nf, x)

Returns 1 if x is an ideal in the number field nf, 0 otherwise.

nfisincl(f, g, flag)

Let f and g define number fields, where f and g are irreducible polynomials in $\mathbb{Q}[X]$ and nf structures as output by nfinit. Tests whether the number field f is conjugate to a subfield of the field g. If they are not, the output is the integer 0. If they are, the output is a vector of polynomials (flag = 0, default) or a single polynomial flag = 1, each polynomial a representing an embedding i.e. being such that g || foa. If either f or g is not irreducible, the result is undefined.

```
? T = x^6 + 3x^4 - 6x^3 + 3x^2 + 18x + 10:
? U = x^3 + 3^*x^2 + 3^*x - 2
? v = nfisincl(U, T);
\%2 = \left[\frac{24}{179} \times 5 - \frac{27}{179} \times 4 + \frac{80}{179} \times 3 - \frac{234}{179} \times 2 + \frac{380}{179} \times 94/179\right]
? subst(U, x, Mod(v[1],T))
\%3 = Mod(0, x^{6} + 3*x^{4} - 6*x^{3} + 3*x^{2} + 18*x + 10)
? #nfisincl(x^2+1, T) \\ two embeddings
\%4 = 2
\\ same result with nf structures
? nfisincl(U, L = nfinit(T)) == v
\%5 = 1
? nfisincl(K = nfinit(U), T) == v
\%6 = 1
? nfisincl(K, L) == v
\%7 = 1
\setminus comparative bench: an nf is a little faster, esp. for the subfield
? B = 10^{3};
? for (i=1, B, nfisincl(U,T))
time = 712 \text{ ms.}
? for (i=1, B, nfisincl(K,T))
time = 485 \text{ ms.}
? for (i=1, B, nfisincl(U,L))
time = 704 \text{ ms.}
? for (i=1, B, nfisincl(K,L))
time = 465 \text{ ms}.
```

Using an *nf* structure for the potential subfield is faster if the structure is already available. On the other hand, the gain in nfisincl is usually not sufficient to make it worthwhile to initialize only for that purpose.

```
? for (i=1, B, nfinit(U))
time = 308 ms.
```

nfisisom(f, g)

As nfisincl, but tests for isomorphism. More efficient if f or g is a number field structure.

```
? f = x^6 + 30*x^5 + 495*x^4 + 1870*x^3 + 16317*x^2 - 22560*x + 59648;
? g = x^6 + 42*x^5 + 999*x^4 + 8966*x^3 + 36117*x^2 + 21768*x + 159332;
? h = x^6 + 30*x^5 + 351*x^4 + 2240*x^3 + 10311*x^2 + 35466*x + 58321;
? #nfisisom(f,g) \\ two isomorphisms
%3 = 2
? nfisisom(f,h) \\ not isomorphic
%4 = 0
\\ comparative bench
? K = nfinit(f); L = nfinit(g); B = 10^3;
? for (i=1, B, nfisisom(f,g))
```

```
time = 6,124 ms.
? for (i=1, B, nfisisom(K,g))
time = 3,356 ms.
? for (i=1, B, nfisisom(f,L))
time = 3,204 ms.
? for (i=1, B, nfisisom(K,L))
time = 3,173 ms.
```

The function is usually very fast when the fields are nonisomorphic, whenever the fields can be distinguished via a simple invariant such as degree, signature or discriminant. It may be slower when the fields share all invariants, but still faster than computing actual isomorphisms:

```
\\ usually very fast when the answer is 'no':
? for (i=1, B, nfisisom(f,h))
time = 32 \text{ ms}.
\\ but not always
2 u = x^{6} + 12x^{5} + 6x^{4} - 377x^{3} - 714x^{2} + 5304x + 15379
? v = x^{6} + 12^{*}x^{5} + 60^{*}x^{4} + 166^{*}x^{3} + 708^{*}x^{2} + 6600^{*}x + 23353
? nfisisom(u,v)
%13 = 0
? polsturm(u) == polsturm(v)
\%14 = 1
? nfdisc(u) == nfdisc(v)
%15 = 1
? for(i=1,B, nfisisom(u,v))
time = 1,821 ms.
? K = nfinit(u); L = nfinit(v);
? for(i=1,B, nfisisom(K,v))
time = 232 \text{ ms.}
```

nfislocalpower(nf, pr, a, n)

Let *nf* be a *nf* structure attached to a number field K, let $a \in K$ and let *pr* be a *prid* structure attached to a maximal ideal v. Return 1 if a is an *n*-th power in the completed local field K_v , and 0 otherwise.

```
? K = nfinit(y^2+1);
? P = idealprimedec(K,2)[1]; \\ the ramified prime above 2
? nfislocalpower(K,P,-1, 2) \\ -1 is a square
%3 = 1
? nfislocalpower(K,P,-1, 4) \\ ... but not a 4-th power
%4 = 0
? nfislocalpower(K,P,2, 2) \\ 2 is not a square
%5 = 0
? Q = idealprimedec(K,5)[1]; \\ a prime above 5
? nfislocalpower(K,Q, [0, 32]~, 30) \\ 32*I is locally a 30-th power
%7 = 1
```

nfkermodpr(nf, x, pr)

This function is obsolete, use nfmodpr.

Kernel of the matrix a in \mathbb{Z}_K/pr , where pr is in **modpr** format (see nfmodprinit).

nfmodpr(*nf*, *x*, *pr*)

Map x to a t_FFELT in the residue field modulo pr. The argument pr is either a maximal ideal in idealprimedec format or, preferably, a *modpr* structure from nfmodprinit. The function nfmodprlift allows to lift back to \mathbb{Z}_K .

Note that the function applies to number field elements and not to vector / matrices / polynomials of such. Use apply to convert recursive structures.

```
? K = nfinit(y^3-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K, P, 't);
? K.zk
%4 = [1, 1/5*y, 1/25*y^2]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, t, 2*t + 1]
? %[1].mod
%6 = t^2 + 3*t + 4
? K.index
%7 = 125
```

For clarity, we represent elements in the residue field $\mathbb{F}_5[t]/(T)$ as polynomials in the variable t. Whenever the underlying rational prime does not divide K.index, it is actually the case that t is the reduction of y in $\mathbb{Q}[y]/(K.pol)$ modulo an irreducible factor of K.pol over \mathbb{F}_p . In the above example, 5 divides the index and t is actually the reduction of y/5.

nfmodprinit(nf, pr, v)

Transforms the prime ideal pr into modpr format necessary for all operations modulo pr in the number field nf. The functions nfmodpr and nfmodprlift allow to project to and lift from the residue field. The variable v is used to display finite field elements (see ffgen).

```
? K = nfinit(y^3-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K, P, 't);
? K.zk
%4 = [1, 1/5*y, 1/25*y^2]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, t, 2*t + 1]
? %[1].mod
%6 = t^2 + 3*t + 4
? K.index
%7 = 125
```

For clarity, we represent elements in the residue field $\mathbb{F}_5[t]/(T)$ as polynomials in the variable t. Whenever the underlying rational prime does not divide K.index, it is actually the case that t is the reduction of y in $\mathbb{Q}[y]/(K.pol)$ modulo an irreducible factor of K.pol over \mathbb{F}_p . In the above example, 5 divides the index and t is actually the reduction of y/5.

nfmodprlift(*nf*, *x*, *pr*)

Lift the t_FFELT x (from nfmodpr) in the residue field modulo pr to the ring of integers. Vectors and matrices are also supported. For polynomials, use apply and the present function.

The argument pr is either a maximal ideal in idealprimedec format or, preferably, a *modpr* structure from nfmodprinit. There are no compatibility checks to try and decide whether x is attached the same residue field as defined by pr: the result is undefined if not.

The function nfmodpr allows to reduce to the residue field.

```
? K = nfinit(y^3-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K,P);
? K.zk
%4 = [1, 1/5*y, 1/25*y^2]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, y, 2*y + 1]
? nfmodprlift(K, %, modP)
%6 = [1, 1/5*y, 2/5*y + 1]
? nfeltval(K, %[3] - K.zk[3], P)
%7 = 1
```

nfnewprec(nf, precision)

Transforms the number field nf into the corresponding data using current (usually larger) precision. This function works as expected if nf is in fact a bnf or a bnr (update structure to current precision). If the original bnf structure was *not* computed by bnfinit(,1), then this may be quite slow and even fail: many generators of principal ideals have to be computed and the algorithm may fail because the accuracy is not sufficient to bootstrap the required generators and fundamental units.

nfpolsturm(*nf*, *T*, *pl*)

Given a polynomial T with coefficients in the number field nf, returns the number of real roots of the s(T) where s runs through the real embeddings of the field specified by optional argument pl:

- pl omitted: all r_1 real places;
- *pl* an integer between 1 and *r*₁: the embedding attached to the *i*-th real root of nf.pol, i.e. nf. roots:math:`[i]`;
- pl a vector or t_VECSMALL: the embeddings attached to the pl[i]-th real roots of nf.pol.

```
? nf = nfinit('y^2 - 2);
? nf.sign
\%2 = [2, 0]
? nf.roots
\%3 = [-1.414..., 1.414...]
? T = x^2 + y;
? nfpolsturm(nf, T, 1) \\ subst(T,y,sqrt(2)) has two real roots
\%5 = 2
? nfpolsturm(nf, T, 2) \setminus subst(T,y,-sqrt(2)) has no real root
\%6 = 0
? nfpolsturm(nf, T) \\ all embeddings together
\%7 = [2, 0]
? nfpolsturm(nf, T, [2,1]) \\ second then first embedding
\%8 = [0, 2]
? nfpolsturm(nf, x^3) \\ number of distinct roots !
\%9 = [1, 1]
? nfpolsturm(nf, x, 6) \setminus there are only 2 real embeddings !
 *** at top-level: nfpolsturm(nf,x,6)
 *** ^_____
 *** nfpolsturm: domain error in nfpolsturm: index > 2
```

nfroots(nf, x)

Roots of the polynomial x in the number field nf given by nfinit without multiplicity (in \mathbb{Q} if nf is omitted). x has coefficients in the number field (scalar, polmod, polynomial, column vector). The main variable of nf must be of lower priority than that of x (see priority (in the PARI manual)). However if the coefficients of the

number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see nffactor).

It is possible to input a defining polynomial for *nf* instead, but this is in general less efficient since parts of an nf structure will then be computed internally. This is useful in two situations: when you do not need the nf elsewhere, or when you cannot initialize an nf due to integer factorization difficulties when attempting to compute the field discriminant and maximal order.

Caveat. nfinit([T, listP]) allows to compute in polynomial time a conditional *nf* structure, which sets nf. zk to an order which is not guaranteed to be maximal at all primes. Always either use nfcertify first (which may not run in polynomial time) or make sure to input nf.pol instead of the conditional *nf*: nfroots is able to recover in polynomial time in this case, instead of potentially missing a factor.

nfrootsof1(nf)

Returns a two-component vector [w, z] where w is the number of roots of unity in the number field nf, and z is a primitive w-th root of unity. It is possible to input a defining polynomial for nf instead.

? K = nfinit(polcyclo(11)); ? nfrootsof1(K) %2 = [22, [0, 0, 0, 0, 0, -1, 0, 0, 0, 0]~] ? z = nfbasistoalg(K, %[2]) \\ in algebraic form %3 = Mod(-x^5, x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1) ? [lift(z^11), lift(z^2)] \\ proves that the order of z is 22 %4 = [-1, -x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 - x^2 - x - 1]

This function guesses the number w as the gcd of the $\#k(v)^*$ for unramified v above odd primes, then computes the roots in *nf* of the *w*-th cyclotomic polynomial. The algorithm is polynomial time with respect to the field degree and the bitsize of the multiplication table in *nf* (both of them polynomially bounded in terms of the size of the discriminant). Fields of degree up to 100 or so should require less than one minute.

nfsnf(nf, x, flag)

Given a torsion \mathbb{Z}_K -module x attached to the square integral invertible pseudo-matrix (A, I, J), returns an ideal list $D = [d_1, ..., d_n]$ which is the Smith normal form of x. In other words, x is isomorphic to $\mathbb{Z}_K/d_1 \oplus ... \oplus \mathbb{Z}_K/d_n$ and d_i divides d_{i-1} for $i \ge 2$. If flag is nonzero return [D, U, V], where UAV is the identity.

See ZKmodules (in the PARI manual) for the definition of integral pseudo-matrix; briefly, it is input as a 3component row vector [A, I, J] where $I = [b_1, ..., b_n]$ and $J = [a_1, ..., a_n]$ are two ideal lists, and A is a square nxn matrix with columns $(A_1, ..., A_n)$, seen as elements in K^n (with canonical basis $(e_1, ..., e_n)$). This data defines the \mathbb{Z}_K module x given by

 $(b_1e_1 \oplus \ldots \oplus b_ne_n)/(a_1A_1 \oplus \ldots \oplus a_nA_n),$

The integrality condition is $a_{i,j} \in b_i a_j^{-1}$ for all i, j. If it is not satisfied, then the d_i will not be integral. Note that every finitely generated torsion module is isomorphic to a module of this form and even with $b_i = Z_K$ for all i.

nfsolvemodpr(*nf*, *a*, *b*, *P*)

This function is obsolete, use nfmodpr.

Let P be a prime ideal in **modpr** format (see nfmodprinit), let a be a matrix, invertible over the residue field, and let b be a column vector or matrix. This function returns a solution of $a \cdot x = b$; the coefficients of x are lifted to *nf* elements.

```
? K = nfinit(y^2+1);
? P = idealprimedec(K, 3)[1];
? P = nfmodprinit(K, P);
? a = [y+1, y; y, 0]; b = [1, y]~
? nfsolvemodpr(K, a,b, P)
%5 = [1, 2]~
```

nfsplitting(*P*, *d*)

Defining polynomial over \mathbb{Q} for the splitting field of $P \in \mathbb{Q}[x]$, that is the smallest field over which P is totally split. If irreducible, the polynomial P can also be given by a nf structure, which is more efficient. If d is given, it must be a multiple of the splitting field degree. Note that if P is reducible the splitting field degree can be smaller than the degree of P.

```
? K = nfinit(x^3-2);
? nfsplitting(K)
%2 = x^6 + 108
? nfsplitting(x^8-2)
%3 = x^16 + 272*x^8 + 64
? S = nfsplitting(x^6-8) // reducible
%4 = x^4+2*x^2+4
? lift(nfroots(subst(S,x,a),x^6-8))
%5 = [-a,a,-1/2*a^3-a,-1/2*a^3,1/2*a^3+a]
```

Specifying the degree of the splitting field can make the computation faster.

```
? nfsplitting(x^17-123);
time = 3,607 ms.
? poldegree(%)
%2 = 272
? nfsplitting(x^17-123,272);
time = 150 ms.
? nfsplitting(x^17-123,273);
*** nfsplitting: Warning: ignoring incorrect degree bound 273
time = 3,611 ms.
```

The complexity of the algorithm is polynomial in the degree d of the splitting field and the bitsize of T; if d is large the result will likely be unusable, e.g. nfinit will not be an option:

```
? nfsplitting(x^6-x-1)
[... degree 720 polynomial deleted ...]
time = 11,020 ms.
```

nfsubfields(pol, d, fl)

Finds all subfields of degree d of the number field defined by the (monic, integral) polynomial *pol* (all subfields if d is null or omitted). The result is a vector of subfields, each being given by [g, h] (default) or simply g (*flag* = 1), where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf. This routine uses

- Allombert's galoissubfields when nf is Galois (with weakly supersolvable Galois group).
- Klüners's or van Hoeij-Klüners-Novocin algorithm in the general case. The latter runs in polynomial time and is generally superior unless there exists a small unramified prime p such that pol has few irreducible factors modulo p.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, expressed as a famat of polynomials with coefficients in the variable of nf, in which case the van Hoeij-Klüners-Novocin algorithm is used.

```
? pol = x^4 - x^3 - x^2 + x + 1;
? nfsubfields(pol)
%2 = [[x, 0], [x^2 - x + 1, x^3 - x^2 + 1], [x^4 - x^3 - x^2 + x + 1, x]]
? nfsubfields(pol,,1)
```

```
%2 = [x, x^2 - x + 1, x^4 - x^3 - x^2 + x + 1]
? y=varhigher("y"); fa = nffactor(pol,subst(pol,x,y));
? #nfsubfields([pol,fa])
%5 = 3
```

nfsubfieldscm(nf, fl)

Compute the maximal CM subfield of *nf*. Return 0 if *nf* does not have a CM subfield, otherwise return [g, h] (default) or g (flag = 1) where g is an absolute equation and h expresses a root of g in terms of the generator of *nf*. Moreover, the CM involution is given by Xmodg(X) : -- - > -Xmodg(X), i.e. Xmodg(X) is a totally imaginary element.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, and *nf* is also allowed to be a monic defining polynomial for the number field.

```
? nf = nfinit(x^8 + 20*x^6 + 10*x^4 - 4*x^2 + 9);
? nfsubfieldscm(nf)
%2 = [x^4 + 4480*x^2 + 3612672, 3*x^5 + 58*x^3 + 5*x]
? pol = y^16-8*y^14+29*y^12-60*y^10+74*y^8-48*y^6+8*y^4+4*y^2+1;
? fa = nffactor(pol, subst(pol,y,x));
? nfsubfieldscm([pol,fa])
%5 = [y^8 + ..., ...]
```

nfsubfieldsmax(nf, fl)

Compute the list of maximal subfields of *nf*. The result is a vector as in nfsubfields.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, and *nf* is also allowed to be a monic defining polynomial for the number field.

norm(x)

Algebraic norm of x, i.e. the product of x with its conjugate (no square roots are taken), or conjugates for polmods. For vectors and matrices, the norm is taken componentwise and hence is not the L^2 -norm (see norm12). Note that the norm of an element of \mathbb{R} is its square, so as to be compatible with the complex norm.

norml2(x)

Square of the L^2 -norm of x. More precisely, if x is a scalar, norml2(x) is defined to be the square of the complex modulus of x (real t_QUAD s are not supported). If x is a polynomial, a (row or column) vector or a matrix, norml2(:math:`x)` is defined recursively as $\sum_i norml2(x_i)$, where (x_i) run through the components of x. In particular, this yields the usual $\sum ||x_i||^2$ (resp. $\sum ||x_{i,j}||^2$) if x is a polynomial or vector (resp. matrix) with complex components.

```
? norml2( [ 1, 2, 3 ] ) \\ vector
%1 = 14
? norml2( [ 1, 2; 3, 4] ) \\ matrix
%2 = 30
? norml2( 2*I + x )
%3 = 5
? norml2( [ [1,2], [3,4], 5, 6 ] ) \\ recursively defined
%4 = 91
```

normlp(*x*, *p*, *precision*)

 L^p -norm of x; sup norm if p is omitted or +00. More precisely, if x is a scalar, normlp(x, p) is defined to be abs(x). If x is a polynomial, a (row or column) vector or a matrix:

• if p is omitted or +oo, then normlp(:math:`x)` is defined recursively as $\max_i normlp(x_i)$), where (x_i)
run through the components of x. In particular, this yields the usual sup norm if x is a polynomial or vector with complex components.

• otherwise, normlp(:math:`x, p)` is defined recursively as $(\sum_i normlp^p(x_i, p))^{1/p}$. In particular, this yields the usual $(\sum ||x_i||^p)^{1/p}$ if x is a polynomial or vector with complex components.

```
? v = [1,-2,3]; normlp(v) \\ vector
\%1 = 3
? normlp(v, +oo) \setminus same, more explicit
\%2 = 3
? M = [1,-2;-3,4]; normlp(M) \\ matrix
\%3 = 4
? T = (1+I) + I*x^2; normlp(T)
%4 = 1.4142135623730950488016887242096980786
? normlp([[1,2], [3,4], 5, 6]) \\ recursively defined
\%5 = 6
? normlp(v, 1)
\%6 = 6
? normlp(M, 1)
\%7 = 10
? normlp(T, 1)
%8 = 2.4142135623730950488016887242096980786
```

numbpart(n)

Gives the number of unrestricted partitions of n, usually called p(n) in the literature; in other words the number of nonnegative integer solutions to a + 2b + 3c + ... = n. n must be of type integer and $n < 10^{15}$ (with trivial values p(n) = 0 for n < 0 and p(0) = 1). The algorithm uses the Hardy-Ramanujan-Rademacher formula. To explicitly enumerate them, see partitions.

numdiv(x)

Number of divisors of ||x||. x must be of type integer.

numerator(f, D)

Numerator of f. This is defined as f * denominator(f, D), see denominator for details. The optional argument D allows to control over which ring we compute the denominator:

- 1: we only consider the underlying Q-structure and the denominator is a (positive) rational integer
- a simple variable, say 'x: all entries as rational functions in K(x) and the denominator is a polynomial in x.

```
? f = x + 1/y + 1/2;
? numerator(f) \\ a t_POL in x
%2 = x + ((y + 2)/(2*y))
? numerator(f, 1) \\ Q-denominator is 2
%3 = x + ((y + 2)/y)
? numerator(f, y) \\ as a rational function in y
%5 = 2*y*x + (y + 2)
```

numtoperm(n, k)

Generates the k-th permutation (as a row vector of length n) of the numbers 1 to n. The number k is taken modulo n!, i.e. inverse function of permutnum. The numbering used is the standard lexicographic ordering, starting at 0.

omega(x)

Number of distinct prime divisors of ||x||. x must be of type integer.

? factor(392)
%1 =
[2 3]
[7 2]
? omega(392)
%2 = 2; \\ without multiplicity
? bigomega(392)
%3 = 5; \\ = 3+2, with multiplicity

00()

Returns an object meaning +oo, for use in functions such as intrum. It can be negated (-oo represents -oo), and compared to real numbers (t_INT, t_FRAC, t_REAL), with the expected meaning: +oo is greater than any real number and -oo is smaller.

padicappr(pol, a)

Vector of *p*-adic roots of the polynomial *pol* congruent to the *p*-adic number *a* modulo *p*, and with the same *p*-adic precision as *a*. The number *a* can be an ordinary *p*-adic number (type t_PADIC, i.e. an element of \mathbb{Z}_p) or can be an integral element of a finite *unramified* extension $\mathbb{Q}_p[X]/(T)$ of \mathbb{Q}_p , given as a t_POLMOD Mod(*A*, *T*) at least one of whose coefficients is a t_PADIC and *T* irreducible modulo *p*. In this case, the result is the vector of roots belonging to the same extension of \mathbb{Q}_p as *a*. The polynomial *pol* should have exact coefficients; if not, its coefficients are first rounded to \mathbb{Q} or $\mathbb{Q}[X]/(T)$ and this is the polynomial whose roots we consider.

padicfields(p, N, flag)

Returns a vector of polynomials generating all the extensions of degree N of the field \mathbb{Q}_p of p-adic rational numbers; N is allowed to be a 2-component vector [n, d], in which case we return the extensions of degree n and discriminant p^d .

The list is minimal in the sense that two different polynomials generate nonisomorphic extensions; in particular, the number of polynomials is the number of classes of nonisomorphic extensions. If P is a polynomial in this list, α is any root of P and $K = \mathbb{Q}_p(\alpha)$, then α is the sum of a uniformizer and a (lift of a) generator of the residue field of K; in particular, the powers of α generate the ring of p-adic integers of K.

If flag = 1, replace each polynomial P by a vector [P, e, f, d, c] where e is the ramification index, f the residual degree, d the valuation of the discriminant, and c the number of conjugate fields. If flag = 2, only return the *number* of extensions in a fixed algebraic closure (Krasner's formula), which is much faster.

padicprec(x, p)

Returns the absolute *p*-adic precision of the object x; this is the minimum precision of the components of x. The result is +00 if x is an exact object (as a *p*-adic):

? padicprec((1 + 0(2^5)) * x + (2 + 0(2^4)), 2) %1 = 4 ? padicprec(x + 2, 2) %2 = +oo ? padicprec(2 + x + 0(x^2), 2) %3 = +oo

The function raises an exception if it encounters an object incompatible with *p*-adic computations:

```
? padicprec(0(3), 2)
 *** at top-level: padicprec(0(3),2)
 *** ^-----
 *** padicprec: inconsistent moduli in padicprec: 3 != 2
```

(continues on next page)

(continued from previous page)

```
? padicprec(1.0, 2)
 *** at top-level: padicprec(1.0,2)
 *** ^_____
*** padicprec: incorrect type in padicprec (t_REAL).
```

parapply(f, x)

Parallel evaluation of f on the elements of x. The function f must not access global variables or variables declared with local(), and must be free of side effects.

```
parapply(factor,[2^256 + 1, 2^193 - 1])
```

factors $2^{256} + 1$ and $2^{193} - 1$ in parallel.

```
{
  my(E = ellinit([1,3]), V = vector(12,i,randomprime(2^200)));
  parapply(p->ellcard(E,p), V)
}
```

computes the order of $E(\mathbb{F}_p)$ for 12 random primes of 200 bits.

pareval(x)

Parallel evaluation of the elements of x, where x is a vector of closures. The closures must be of arity 0, must not access global variables or variables declared with local and must be free of side effects.

Here is an artificial example explaining the MOV attack on the elliptic discrete log problem (by reducing it to a standard discrete log over a finite field):

```
{
    my(q = 2^30 + 3, m = 40 * q; p = 1 + m^2); \\ p, q are primes
    my(E = ellinit([0,0,0,1,0] * Mod(1,p)));
    my([P, Q] = ellgenerators(E));
    \\ E(F_p) ~ Z/m P + Z/m Q and the order of the
    \\ Weil pairing <P,Q> in (Z/p)^* is m
    my(F = [m,factor(m)], e = random(m), R, wR, wQ);
    R = ellpow(E, Q, e);
    wR = ellweilpairing(E,P,R,m);
    wQ = ellweilpairing(E,P,Q,m); \\ wR = wQ^e
    pareval([()->znlog(wR,wQ,F), ()->elllog(E,R,Q), ()->e])
}
```

Note the use of my to pass "arguments" to the functions we need to evaluate while satisfying the listed requirements: closures of arity 0 and no global variables (another possibility would be to use export). As a result, the final three statements satisfy all the listed requirements and are run in parallel. (Which is silly for this computation but illustrates the use of pareval.) The function parfor is more powerful but harder to use.

parselect(f, A, flag)

Selects elements of A according to the selection function f, done in parallel. If *flag* is 1, return the indices of those elements (indirect selection) The function f must not access global variables or variables declared with local(), and must be free of side effects.

partitions(k, a, n)

Returns the vector of partitions of the integer k as a sum of positive integers (parts); for k < 0, it returns the empty set [], and for k = 0 the trivial partition (no parts). A partition is given by a t_VECSMALL, where parts are sorted in nondecreasing order:

? partitions(3)
%1 = [Vecsmall([3]), Vecsmall([1, 2]), Vecsmall([1, 1, 1])]

correspond to 3, 1 + 2 and 1 + 1 + 1. The number of (unrestricted) partitions of k is given by numbpart:

```
? #partitions(50)
%1 = 204226
? numbpart(50)
%2 = 204226
```

Optional parameters n and a are as follows:

- n = nmax (resp. n = [nmin, nmax]) restricts partitions to length less than nmax (resp. length between nmin and nmax), where the *length* is the number of nonzero entries.
- a = amax (resp. a = [amin, amax]) restricts the parts to integers less than amax (resp. between amin and amax).

```
? partitions(4, 2) \\ parts bounded by 2
%1 = [Vecsmall([2, 2]), Vecsmall([1, 1, 2]), Vecsmall([1, 1, 1, 1])]
? partitions(4,, 2) \\ at most 2 parts
%2 = [Vecsmall([4]), Vecsmall([1, 3]), Vecsmall([2, 2])]
? partitions(4,[0,3], 2) \\ at most 2 parts
%3 = [Vecsmall([4]), Vecsmall([1, 3]), Vecsmall([2, 2])]
```

By default, parts are positive and we remove zero entries unless $amin \le 0$, in which case nmin is ignored and we fix #X = nmax:

```
? partitions(4, [0,3]) \\ parts between 0 and 3
%1 = [Vecsmall([0, 0, 1, 3]), Vecsmall([0, 0, 2, 2]),\
Vecsmall([0, 1, 1, 2]), Vecsmall([1, 1, 1, 1])]
? partitions(1, [0,3], [2,4]) \\ no partition with 2 to 4 nonzero parts
%2 = []
```

permcycles(x)

Given a permutation x on n elements, return the orbits of 1, ..., n under the action of x as cycles.

```
? permcycles(Vecsmall([1,2,3]))
%1 = [Vecsmall([1]),Vecsmall([2]),Vecsmall([3])]
? permcycles(Vecsmall([2,3,1]))
%2 = [Vecsmall([1,2,3])]
? permcycles(Vecsmall([2,1,3]))
%3 = [Vecsmall([1,2]),Vecsmall([3])]
```

permorder(x)

Given a permutation x on n elements, return its order.

```
? p = Vecsmall([3,1,4,2,5]);
? p^2
%2 = Vecsmall([4,3,2,1,5])
? p^4
%3 = Vecsmall([1,2,3,4,5])
? permorder(p)
%4 = 4
```

permsign(x)

Given a permutation x on n elements, return its signature.

```
? p = Vecsmall([3,1,4,2,5]);
? permsign(p)
%2 = -1
? permsign(p^2)
%3 = 1
```

permtonum(x)

Given a permutation x on n elements, gives the number k such that x = numtoperm(n, k), i.e. inverse function of numtoperm. The numbering used is the standard lexicographic ordering, starting at 0.

plotbox(w, x2, y2, filled)

Let (x1, y1) be the current position of the virtual cursor. Draw in the rectwindow w the outline of the rectangle which is such that the points (x1, y1) and (x2, y2) are opposite corners. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move. If filled = 1, fill the box.

plotclip(w)

clips' the content of rectwindow :math: w, i.e remove all parts of the drawing that would not be visible on the screen. Together with plotcopy this function enables you to draw on a scratchpad before committing the part you're interested in to the final picture.

plotcolor(w, c)

Set default color to c in rectwindow w. Return [R,G,B] value attached to color. Possible values for c are

- a t_VEC or t_VECSMALL [R, G, B] giving the color RGB value (all 3 values are between 0 and 255), e.g. [250,235,215] or equivalently [0xfa, 0xeb, 0xd7] for antiquewhite;
- a t_STR giving a valid colour name (see the rgb.txt file in X11 distributions), e.g. "antiquewhite" or an RGV value given by a # followed by 6 hexadecimal digits, e.g. "#faebd7" for antiquewhite;
- a t_INT, an index in the graphcolormap default, factory setting are
- 1 = black, 2 = blue, 3 = violetred, 4 = red, 5 = green, 6 = grey, 7 = gainsborough.

but this can be extended if needed.

```
? plotinit(0,100,100);
? plotcolor(0, "turquoise")
%2 = [64, 224, 208]
? plotbox(0, 50,50,1);
? plotmove(0, 50,50);
? plotcolor(0, 2) \\ blue
%4 = [0, 0, 255]
? plotbox(0, 50,50,1);
? plotdraw(0);
```

plotcopy(sourcew, destw, dx, dy, flag)

Copy the contents of rectwindow *sourcew* to rectwindow *destw* with offset (dx,dy). If flag's bit 1 is set, dx and dy express fractions of the size of the current output device, otherwise dx and dy are in pixels. dx and dy are relative positions of northwest corners if other bits of flag vanish, otherwise of: 2: southwest, 4: southeast, 6: northeast corners

plotcursor(w)

Give as a 2-component vector the current (scaled) position of the virtual cursor corresponding to the rectwindow w.

plotdraw(w, flag)

Physically draw the rectwindow w. More generally, w can be of the form $[w_1, x_1, y_1, w_2, x_2, y_2, ...]$ (number of components must be divisible by 3; the windows w_1, w_2 , etc. are physically placed with their upper left corner at physical position $(x_1, y_1), (x_2, y_2), ...$ respectively, and are then drawn together. Overlapping regions will thus be drawn twice, and the windows are considered transparent. Then display the whole drawing in a window on your screen. If $flag! = 0, x_1, y_1$ etc. express fractions of the size of the current output device

plotexport(fmt, list, flag)

Draw list of rectwindows as in plotdraw(list,flag), returning the resulting picture as a character string which can then be written to a file. The format fmt is either "ps" (PostScript output) or "svg" (Scalable Vector Graphics).

? plotinit(0, 100, 100); ? plotbox(0, 50, 50); ? plotcolor(0, 2); ? plotbox(0, 30, 30); ? plotdraw(0); \\ watch result on screen ? s = plotexport("svg", 0); ? write("graph.svg", s); \\ dump result to file

plothraw(X, Y, flag)

Given X and Y two vectors of equal length, plots (in high precision) the points whose (x, y)-coordinates are given in X and Y. Automatic positioning and scaling is done, but with the same scaling factor on x and y. If flag is 1, join points, other nonzero flags toggle display options and should be combinations of bits 2^k , $k \ge 3$ as in ploth.

plothrawexport(fmt, X, Y, flag)

Given X and Y two vectors of equal length, plots (in high precision) the points whose (x, y)-coordinates are given in X and Y, returning the resulting picture as a character string which can then be written to a file. The format fmt is either "ps" (PostScript output) or "svg" (Scalable Vector Graphics).

Automatic positioning and scaling is done, but with the same scaling factor on x and y. If flag is 1, join points, other nonzero flags toggle display options and should be combinations of bits 2^k , $k \ge 3$ as in ploth.

plothsizes(flag)

Return data corresponding to the output window in the form of a 8-component vector: window width and height, sizes for ticks in horizontal and vertical directions (this is intended for the gnuplot interface and is currently not significant), width and height of characters, width and height of display, if applicable. If display has no sense, e.g. for svg plots or postscript plots, then width and height of display are set to 0.

If flag = 0, sizes of ticks and characters are in pixels, otherwise are fractions of the screen size

plotinit(w, x, y, flag)

Initialize the rectwindow w, destroying any rect objects you may have already drawn in w. The virtual cursor is set to (0,0). The rectwindow size is set to width x and height y; omitting either x or y means we use the full size of the device in that direction. If flag = 0, x and y represent pixel units. Otherwise, x and y are understood as fractions of the size of the current output device (hence must be between 0 and 1) and internally converted to pixels.

The plotting device imposes an upper bound for x and y, for instance the number of pixels for screen output. These bounds are available through the plothsizes function. The following sequence initializes in a portable way (i.e independent of the output device) a window of maximal size, accessed through coordinates in the [0, 1000]x[0, 1000] range:

```
s = plothsizes();
plotinit(0, s[1]-1, s[2]-1);
plotscale(0, 0,1000, 0,1000);
```

plotkill(w)

Erase rectwindow w and free the corresponding memory. Note that if you want to use the rectwindow w again, you have to use plotinit first to specify the new size. So it's better in this case to use plotinit directly as this throws away any previous work in the given rectwindow.

plotlines(w, X, Y, flag)

Draw on the rectwindow w the polygon such that the (x,y)-coordinates of the vertices are in the vectors of equal length X and Y. For simplicity, the whole polygon is drawn, not only the part of the polygon which is inside the rectwindow. If flag is nonzero, close the polygon. In any case, the virtual cursor does not move.

X and Y are allowed to be scalars (in this case, both have to). There, a single segment will be drawn, between the virtual cursor current position and the point (X, Y). And only the part thereof which actually lies within the boundary of w. Then *move* the virtual cursor to (X, Y), even if it is outside the window. If you want to draw a line from (x1, y1) to (x2, y2) where (x1, y1) is not necessarily the position of the virtual cursor, use plotmove(w,x1,y1) before using this function.

plotlinetype(w, type)

This function is obsolete and currently a no-op.

Change the type of lines subsequently plotted in rectwindow w. type -2 corresponds to frames, -1 to axes, larger values may correspond to something else. w = -1 changes highlevel plotting.

plotmove(w, x, y)

Move the virtual cursor of the rectwindow w to position (x, y).

plotpoints(w, X, Y)

Draw on the rectwindow w the points whose (x, y)-coordinates are in the vectors of equal length X and Y and which are inside w. The virtual cursor does *not* move. This is basically the same function as plothraw, but either with no scaling factor or with a scale chosen using the function plotscale.

As was the case with the plotlines function, X and Y are allowed to be (simultaneously) scalar. In this case, draw the single point (X, Y) on the rectwindow w (if it is actually inside w), and in any case move the virtual cursor to position (x, y).

If you draw few points in the rectwindow, they will be hard to see; in this case, you can use filled boxes instead. Compare:

```
? plotinit(0, 100,100); plotpoints(0, 50,50);
? plotdraw(0)
? plotinit(1, 100,100); plotmove(1,48,48); plotrbox(1, 4,4, 1);
? plotdraw(1)
```

plotpointsize(w, size)

This function is obsolete. It is currently a no-op.

Changes the "size" of following points in rectwindow w. If w = -1, change it in all rectwindows.

plotpointtype(w, type)

This function is obsolete and currently a no-op.

change the type of points subsequently plotted in rectwindow w. type = -1 corresponds to a dot, larger values may correspond to something else. w = -1 changes highlevel plotting.

plotrbox(w, dx, dy, filled)

Draw in the rectwindow w the outline of the rectangle which is such that the points (x1, y1) and (x1+dx, y1+dy) are opposite corners, where (x1, y1) is the current position of the cursor. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move. If filled = 1, fill the box.

plotrecthraw(w, data, flags)

Plot graph(s) for *data* in rectwindow w; *flag* has the same meaning here as in ploth, though recursive plot is no longer significant.

The argument *data* is a vector of vectors, each corresponding to a list a coordinates. If parametric plot is set, there must be an even number of vectors, each successive pair corresponding to a curve. Otherwise, the first one contains the x coordinates, and the other ones contain the y-coordinates of curves to plot.

plotrline(w, dx, dy)

Draw in the rectwindow w the part of the segment (x1, y1) - (x1+dx, y1+dy) which is inside w, where (x1, y1) is the current position of the virtual cursor, and move the virtual cursor to (x1 + dx, y1 + dy) (even if it is outside the window).

plotrmove(w, dx, dy)

Move the virtual cursor of the rectwindow w to position (x1 + dx, y1 + dy), where (x1, y1) is the initial position of the cursor (i.e. to position (dx, dy) relative to the initial cursor).

plotrpoint(w, dx, dy)

Draw the point (x1 + dx, y1 + dy) on the rectwindow w (if it is inside w), where (x1, y1) is the current position of the cursor, and in any case move the virtual cursor to position (x1 + dx, y1 + dy).

If you draw few points in the rectwindow, they will be hard to see; in this case, you can use filled boxes instead. Compare:

```
? plotinit(0, 100,100); plotrpoint(0, 50,50); plotrpoint(0, 10,10);
? plotdraw(0)
```

```
? thickpoint(w,x,y)= plotmove(w,x-2,y-2); plotrbox(w,4,4,1);
? plotinit(1, 100,100); thickpoint(1, 50,50); thickpoint(1, 60,60);
? plotdraw(1)
```

plotscale(*w*, *x1*, *x2*, *y1*, *y2*)

Scale the local coordinates of the rectwindow w so that x goes from x1 to x2 and y goes from y1 to y2 (x2 < x1 and y2 < y1 being allowed). Initially, after the initialization of the rectwindow w using the function plotinit, the default scaling is the graphic pixel count, and in particular the y axis is oriented downwards since the origin is at the upper left. The function plotscale allows to change all these defaults and should be used whenever functions are graphed.

plotstring(w, x, flags)

Draw on the rectwindow w the String x (see strings (in the PARI manual)), at the current position of the cursor.

flag is used for justification: bits 1 and 2 regulate horizontal alignment: left if 0, right if 2, center if 1. Bits 4 and 8 regulate vertical alignment: bottom if 0, top if 8, v-center if 4. Can insert additional small gap between point and string: horizontal if bit 16 is set, vertical if bit 32 is set (see the tutorial for an example).

polchebyshev(n, flag, a)

Returns the n - th Chebyshev polynomial of the first kind T_n (flag = 1) or the second kind U_n (flag = 2), evaluated at a ('x by default). Both series of polynomials satisfy the 3-term relation

$$P_{n+1} = 2xP_n - P_{n-1},$$

and are determined by the initial conditions $U_0 = T_0 = 1$, $T_1 = x$, $U_1 = 2x$. In fact $T'_n = nU_{n-1}$ and, for all complex numbers z, we have $T_n(\cos z) = \cos(nz)$ and $U_{n-1}(\cos z) = \sin(nz)/\sin z$. If $n \ge 0$, then these polynomials have degree n. For n < 0, T_n is equal to T_{-n} and U_n is equal to $-U_{-2-n}$. In particular, $U_{-1} = 0$.

polclass(D, inv, x)

Return a polynomial in $\mathbb{Z}[x]$ generating the Hilbert class field for the imaginary quadratic discriminant D. If *inv* is 0 (the default), use the modular *j*-function and return the classical Hilbert polynomial, otherwise use a class invariant. The following invariants correspond to the different values of *inv*, where *f* denotes Weber's function weber, and $w_{p,q}$ the double eta quotient given by $w_{p,q} = (\eta(x/p)\eta(x/q))/(\eta(x)\eta(x/pq))$

The invariants $w_{p,q}$ are not allowed unless they satisfy the following technical conditions ensuring they do generate the Hilbert class field and not a strict subfield:

- if p! = q, we need them both noninert, prime to the conductor of $\mathbb{Z}[\sqrt{D}]$. Let P, Q be prime ideals above p and q; if both are unramified, we further require that P^1Q^1 be all distinct in the class group of $\mathbb{Z}[\sqrt{D}]$; if both are ramified, we require that PQ! = 1 in the class group.
- if p = q, we want it split and prime to the conductor and the prime ideal above it must have order ! = 1, 2, 4 in the class group.

Invariants are allowed under the additional conditions on D listed below.

- 0: j
- $1: f, D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- $2: f^2, D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- $3: f^3, D = 1mod8;$
- $4: f^4, D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- $5: \gamma_2 = j^{1/3}, D = 1, 2mod3;$
- $6: w_{2,3}, D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- 8: f^8 , $D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- 9: $w_{3,3}$, $D = 1 \mod 2$ and $D = 1, 2 \mod 3$;
- 10: $w_{2.5}$, $D! = 60 \mod 80$ and $D = 1, 2 \mod 3$;
- 14: $w_{2,7}, D = 1 \mod 8;$
- 15: $w_{3,5}$, D = 1, 2mod3;
- 21: $w_{3,7}$, $D = 1 \mod 2$ and 21 does not divide D
- 23: $w_{2,3}^2$, D = 1, 2mod3;
- 24: $w_{2,5}^2$, D = 1, 2mod3;
- 26: $w_{2,13}, D! = 156 \mod 208;$
- 27: $w_{2,7}^2$, D! = 28mod112;
- 28: $w_{3,3}^2$, D = 1, 2mod3;
- 35: $w_{5,7}$, D = 1, 2mod3;
- 39: $w_{3,13}$, $D = 1 \mod 2$ and $D = 1, 2 \mod 3$;

The algorithm for computing the polynomial does not use the floating point approach, which would evaluate a precise modular function in a precise complex argument. Instead, it relies on a faster Chinese remainder based approach modulo small primes, in which the class invariant is only defined algebraically by the modular polynomial relating the modular function to j. So in fact, any of the several roots of the modular polynomial may actually be the class invariant, and more precise assertions cannot be made.

For instance, while polclass(D) returns the minimal polynomial of $j(\tau)$ with τ (any) quadratic integer for the discriminant D, the polynomial returned by polclass(D, 5) can be the minimal polynomial of any of $\gamma_2(\tau)$, $\zeta_3\gamma_2(\tau)$ or $\zeta_3^2\gamma_2(\tau)$, the three roots of the modular polynomial $j = \gamma_2^3$, in which j has been specialised to $j(\tau)$.

The modular polynomial is given by $j = ((f^{24} - 16)^3)/(f^{24})$ for Weber's function f.

For the double eta quotients of level N = pq, all functions are covered such that the modular curve $X_0^+(N)$, the function field of which is generated by the functions invariant under $\Gamma^0(N)$ and the Fricke-Atkin-Lehner involution, is of genus 0 with function field generated by (a power of) the double eta quotient w. This ensures that the full Hilbert class field (and not a proper subfield) is generated by class invariants from these double eta quotients. Then the modular polynomial is of degree 2 in j, and of degree $\psi(N) = (p+1)(q+1)$ in w.

```
? polclass(-163)
%1 = x + 262537412640768000
? polclass(-51, , 'z)
%2 = z^2 + 5541101568*z + 6262062317568
? polclass(-151,1)
x^7 - x^6 + x^5 + 3*x^3 - x^2 + 3*x + 1
```

polcoef(*x*, *n*, *v*)

Coefficient of degree n of the polynomial x, with respect to the main variable if v is omitted, with respect to v otherwise. If n is greater than the degree, the result is zero.

Naturally applies to scalars (polynomial of degree 0), as well as to rational functions whose denominator is a monomial. It also applies to power series: if n is less than the valuation, the result is zero. If it is greater than the largest significant degree, then an error message is issued.

polcoeff(*x*, *n*, *v*)

Deprecated alias for polcoef.

polcompositum(P, Q, flag)

P and *Q* being squarefree polynomials in $\mathbb{Z}[X]$ in the same variable, outputs the simple factors of the étale \mathbb{Q} -algebra $A = \mathbb{Q}(X,Y)/(P(X),Q(Y))$. The factors are given by a list of polynomials *R* in $\mathbb{Z}[X]$, attached to the number field $\mathbb{Q}(X)/(R)$, and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. In this case, there will be a single factor R if and only if the number fields defined by P and Q are linearly disjoint (their intersection is \mathbb{Q}).

Assuming P is irreducible (of smaller degree than Q for efficiency), it is in general much faster to proceed as follows

```
nf = nfinit(P); L = nffactor(nf, Q)[,1];
vector(#L, i, rnfequation(nf, L[i]))
```

to obtain the same result. If you are only interested in the degrees of the simple factors, the rnfequation instruction can be replaced by a trivial poldegree(P) * poldegree(L[i]).

The binary digits of flag mean

1: outputs a vector of 4-component vectors [R, a, b, k], where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of $\mathbb{Q}(X)/(R)$. Finally, k is a small integer such that b + ka = X modulo R.

2: assume that P and Q define number fields which are linearly disjoint: both polynomials are irreducible and the corresponding number fields have no common subfield besides \mathbb{Q} . This allows to save a costly factorization over \mathbb{Q} . In this case return the single simple factor instead of a vector with one element.

A compositum is often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is an example involving the field $\mathbb{Q}(\zeta_5, 5^{1/5})$:

```
? L = polcompositum(x^5 - 5, polcyclo(5), 1); \\ list of [R,a,b,k]
? [R, a] = L[1]; \setminus pick the single factor, extract R,a (ignore b,k)
? R \\ defines the compositum
3 = x^20 + 5x^{19} + 15x^{18} + 35x^{17} + 70x^{16} + 141x^{15} + 260x^{14}
+ 355*x^13 + 95*x^12 - 1460*x^11 - 3279*x^10 - 3660*x^9 - 2005*x^8 \
+ 705*x^7 + 9210*x^6 + 13506*x^5 + 7145*x^4 - 2740*x^3 + 1040*x^2 \
- 320*x + 256
? a^5 - 5 \setminus a fifth root of 5
%4 = 0
? [T, X] = polredbest(R, 1);
? T \\ simpler defining polynomial for Q[x]/(R)
\%6 = x^20 + 25 x^10 + 5
? X \\ root of R in Q[y]/(T(y))
\%7 = Mod(-1/11*x^{15} - 1/11*x^{14} + 1/22*x^{10} - 47/22*x^{5} - 29/11*x^{4} + 7/22,)
x^{20} + 25 * x^{10} + 5
? a = subst(a.pol, 'x, X) \setminus a in the new coordinates
\%8 = Mod(1/11*x^{14} + 29/11*x^{4}, x^{20} + 25*x^{10} + 5)
? a^5 - 5
\%9 = 0
```

In the above example, $x^5 - 5$ and the 5-th cyclotomic polynomial are irreducible over \mathbb{Q} ; they have coprime degrees so define linearly disjoint extensions and we could have started by

? [R,a] = polcompositum(x^5 - 5, polcyclo(5), 3); \\ [R,a,b,k]

polcyclo(*n*, *a*)

n-th cyclotomic polynomial, evaluated at a ('x by default). The integer n must be positive.

Algorithm used: reduce to the case where n is squarefree; to compute the cyclotomic polynomial, use $\Phi_{np}(x) = \Phi_n(x^p)/\Phi(x)$; to compute it evaluated, use $\Phi_n(x) = \prod_{d \mid n} (x^d - 1)^{\mu(n/d)}$. In the evaluated case, the algorithm assumes that $a^d - 1$ is either 0 or invertible, for all $d \mid n$. If this is not the case (the base ring has zero divisors), use subst(polcyclo(n), x, a).

polcyclofactors(f)

Returns a vector of polynomials, whose product is the product of distinct cyclotomic polynomials dividing f.

```
? f = x^10+5*x^8-x^7+8*x^6-4*x^5+8*x^4-3*x^3+7*x^2+3;
? v = polcyclofactors(f)
%2 = [x^2 + 1, x^2 + x + 1, x^4 - x^3 + x^2 - x + 1]
? apply(poliscycloprod, v)
%3 = [1, 1, 1]
? apply(poliscyclo, v)
%4 = [4, 3, 10]
```

In general, the polynomials are products of cyclotomic polynomials and not themselves irreducible:

```
? g = x^8+2*x^7+6*x^6+9*x^5+12*x^4+11*x^3+10*x^2+6*x+3;
? polcyclofactors(g)
%2 = [x^6 + 2*x^5 + 3*x^4 + 3*x^3 + 3*x^2 + 2*x + 1]
? factor(%[1])
%3 =
```

(continues on next page)

(continued from previous page)

 $[x^{2} + x + 1 1]$ $[x^{4} + x^{3} + x^{2} + x + 1 1]$

poldegree(x, v)

Degree of the polynomial x in the main variable if v is omitted, in the variable v otherwise.

The degree of 0 is $-\infty$. The degree of a nonzero scalar is 0. Finally, when x is a nonzero polynomial or rational function, returns the ordinary degree of x. Raise an error otherwise.

poldisc(pol, v)

Discriminant of the polynomial *pol* in the main variable if v is omitted, in v otherwise. Uses a modular algorithm over \mathbb{Z} or \mathbb{Q} , and the subresultant algorithm otherwise.

? T = x^4 + 2*x+1; ? poldisc(T) %2 = -176 ? poldisc(T^2) %3 = 0

For convenience, the function also applies to types t_QUAD and t_QFI/t_QFR:

```
? z = 3*quadgen(8) + 4;
? poldisc(z)
%2 = 8
? q = Qfb(1,2,3);
? poldisc(q)
%4 = -8
```

poldiscfactors(T, flag)

Given a polynomial T with integer coefficients, return [D, faD] where D is the discriminant of T and faD is a cheap partial factorization of ||D||: entries in its first column are coprime and not perfect powers but need not be primes. The factors are obtained by a combination of trial division, testing for perfect powers, factorizations in coprimes, and computing Euclidean remainder sequences for (T, T') modulo composite factors d of D (which is likely to produce 0-divisors in $\mathbb{Z}/d\mathbb{Z}$). If *flag* is 1, finish the factorization using factorint.

```
? T = x<sup>3</sup> - 6021021*x<sup>2</sup> + 12072210077769*x - 8092423140177664432;
? [D,faD] = poldiscfactors(T); print(faD); D
[3, 3; 7, 2; 373, 2; 500009, 2; 24639061, 2]
%2 = -27937108625866859018515540967767467
? T = x<sup>3</sup> + 9*x<sup>2</sup> + 27*x - 125014250689643346789780229390526092263790263725;
? [D,faD] = poldiscfactors(T); print(faD)
[2, 6; 3, 3; 125007125141751093502187, 4]
? [D,faD] = poldiscfactors(T, 1); print(faD)
[2, 6; 3, 3; 500009, 12; 1000003, 4]
```

poldiscreduced(f)

Reduced discriminant vector of the (integral, monic) polynomial f. This is the vector of elementary divisors of $\mathbb{Z}[\alpha]/f'(\alpha)\mathbb{Z}[\alpha]$, where α is a root of the polynomial f. The components of the result are all positive, and their product is equal to the absolute value of the discriminant of f.

polgalois(T, precision)

Galois group of the nonconstant polynomial $T \in \mathbb{Q}[X]$. In the present version **2.13.3**, T must be irreducible and

the degree d of T must be less than or equal to 7. If the galdata package has been installed, degrees 8, 9, 10 and 11 are also implemented. By definition, if $K = \mathbb{Q}[x]/(T)$, this computes the action of the Galois group of the Galois closure of K on the d distinct roots of T, up to conjugacy (corresponding to different root orderings).

The output is a 4-component vector [n, s, k, name] with the following meaning: n is the cardinality of the group, s is its signature (s = 1 if the group is a subgroup of the alternating group A_d , s = -1 otherwise) and name is a character string containing name of the transitive group according to the GAP 4 transitive groups library by Alexander Hulpke.

k is more arbitrary and the choice made up to version 2.2.3 of PARI is rather unfortunate: for d > 7, k is the numbering of the group among all transitive subgroups of S_d , as given in "The transitive groups of degree up to eleven", G. Butler and J. McKay, *Communications in Algebra*, vol. 11, 1983, pp. 863–911 (group k is denoted T_k there). And for d <= 7, it was ad hoc, so as to ensure that a given triple would denote a unique group. Specifically, for polynomials of degree d <= 7, the groups are coded as follows, using standard notations

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1], S_3 = [6, -1, 1].$

In degree 4: $C_4 = [4, -1, 1], V_4 = [4, 1, 1], D_4 = [8, -1, 1], A_4 = [12, 1, 1], S_4 = [24, -1, 1].$

In degree 5: $C_5 = [5, 1, 1], D_5 = [10, 1, 1], M_{20} = [20, -1, 1], A_5 = [60, 1, 1], S_5 = [120, -1, 1].$

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 1]$, $A_4 = [12, 1, 1]$, $G_{18} = [18, -1, 1]$, $S_4^- = [24, -1, 1]$, $A_4xC_2 = [24, -1, 2]$, $S_4^+ = [24, 1, 1]$, $G_{36}^- = [36, -1, 1]$, $G_{36}^+ = [36, 1, 1]$, $S_4xC_2 = [48, -1, 1]$, $A_5 = PSL_2(5) = [60, 1, 1]$, $G_{72} = [72, -1, 1]$, $S_5 = PGL_2(5) = [120, -1, 1]$, $A_6 = [360, 1, 1]$, $S_6 = [720, -1, 1]$.

In degree 7: $C_7 = [7, 1, 1], D_7 = [14, -1, 1], M_{21} = [21, 1, 1], M_{42} = [42, -1, 1], PSL_2(7) = PSL_3(2) = [168, 1, 1], A_7 = [2520, 1, 1], S_7 = [5040, -1, 1].$

This is deprecated and obsolete, but for reasons of backward compatibility, we cannot change this behavior yet. So you can use the default new_galois_format to switch to a consistent naming scheme, namely k is always the standard numbering of the group among all transitive subgroups of S_n . If this default is in effect, the above groups will be coded as:

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1], S_3 = [6, -1, 2].$

In degree 4: $C_4 = [4, -1, 1], V_4 = [4, 1, 2], D_4 = [8, -1, 3], A_4 = [12, 1, 4], S_4 = [24, -1, 5].$

In degree 5: $C_5 = [5, 1, 1], D_5 = [10, 1, 2], M_{20} = [20, -1, 3], A_5 = [60, 1, 4], S_5 = [120, -1, 5].$

In degree 6: $C_6 = [6, -1, 1], S_3 = [6, -1, 2], D_6 = [12, -1, 3], A_4 = [12, 1, 4], G_{18} = [18, -1, 5], A_4xC_2 = [24, -1, 6], S_4^+ = [24, 1, 7], S_4^- = [24, -1, 8], G_{36}^- = [36, -1, 9], G_{36}^+ = [36, 1, 10], S_4xC_2 = [48, -1, 11], A_5 = PSL_2(5) = [60, 1, 12], G_{72} = [72, -1, 13], S_5 = PGL_2(5) = [120, -1, 14], A_6 = [360, 1, 15], S_6 = [720, -1, 16].$

In degree 7: $C_7 = [7, 1, 1], D_7 = [14, -1, 2], M_{21} = [21, 1, 3], M_{42} = [42, -1, 4], PSL_2(7) = PSL_3(2) = [168, 1, 5], A_7 = [2520, 1, 6], S_7 = [5040, -1, 7].$

Warning. The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

polgraeffe(f)

Returns the Graeffe transform g of f, such that $g(x^2) = f(x)f(-x)$.

polhensellift(A, B, p, e)

Given a prime p, an integral polynomial A whose leading coefficient is a p-unit, a vector B of integral polynomials that are monic and pairwise relatively prime modulo p, and whose product is congruent to A/lc(A) modulo p, lift the elements of B to polynomials whose product is congruent to A modulo p^e .

More generally, if T is an integral polynomial irreducible mod p, and B is a factorization of A over the finite field $\mathbb{F}_p[t]/(T)$, you can lift it to $\mathbb{Z}_p[t]/(T, p^e)$ by replacing the p argument with [p, T]:

? { T = t³ - 2; p = 7; A = x² + t + 1; B = [x + (3*t² + t + 1), x + (4*t² + 6*t + 6)]; r = polhensellift(A, B, [p, T], 6) } %1 = [x + (20191*t² + 50604*t + 75783), x + (97458*t² + 67045*t + 41866)] ? liftall(r[1] * r[2] * Mod(Mod(1,p⁶),T)) %2 = x² + (t + 1)

polhermite(n, a, flag)

n-th Hermite polynomial H_n evaluated at a ('x by default), i.e.

$$H_n(x) = (-1)^n e^{x^2} (d^n) / (dx^n) e^{-x^2}.$$

If *flag* is nonzero and n > 0, return $[H_{n-1}(a), H_n(a)]$.

```
? polhermite(5)
%1 = 32*x^5 - 160*x^3 + 120*x
? polhermite(5, -2) \\ H_5(-2)
%2 = 16
? polhermite(5,,1)
%3 = [16*x^4 - 48*x^2 + 12, 32*x^5 - 160*x^3 + 120*x]
? polhermite(5,-2,1)
%4 = [76, 16]
```

polinterpolate(X, Y, t, e)

Given the data vectors X and Y of the same length n (X containing the x-coordinates, and Y the corresponding y-coordinates), this function finds the interpolating polynomial P of minimal degree passing through these points and evaluates it at t. If Y is omitted, the polynomial P interpolates the (i, X[i]).

```
? v = [1, 2, 4, 8, 11, 13];
? P = polinterpolate(v) \\ formal interpolation
%1 = 7/120*x^5 - 25/24*x^4 + 163/24*x^3 - 467/24*x^2 + 513/20*x - 11
? [ subst(P, 'x,a) | a <- [1..6] ]
%2 = [1, 2, 4, 8, 11, 13]
? polinterpolate(v,, 10) \\ evaluate at 10
%3 = 508
? subst(P, x, 10)
%4 = 508
? P = polinterpolate([1,2,4], [9,8,7])
%5 = 1/6*x^2 - 3/2*x + 31/3
? [subst(P, 'x, a) | a <- [1,2,4]]
%6 = [9, 8, 7]
? P = polinterpolate([1,2,4], [9,8,7], 0)
%7 = 31/3
```

If the goal is to extrapolate a function at a unique point, it is more efficient to use the t argument rather than interpolate formally then evaluate:

```
? x0 = 1.5;
? v = vector(20, i,random([-10,10]));
? for(i=1,10^3, subst(polinterpolate(v),'x, x0))
time = 352 ms.
? for(i=1,10^3, polinterpolate(v,,x0))
time = 111 ms.
? v = vector(40, i,random([-10,10]));
? for(i=1,10^3, subst(polinterpolate(v), 'x, x0))
time = 3,035 ms.
? for(i=1,10^3, polinterpolate(v,, x0))
time = 436 ms.
```

The threshold depends on the base field. Over small prime finite fields, interpolating formally first is more efficient

```
? bench(p, N, T = 10^3) =
{ my (v = vector(N, i, random(Mod(0,p))));
 my (x0 = Mod(3, p), t1, t2);
gettime();
for(i=1, T, subst(polinterpolate(v), 'x, x0));
t1 = gettime();
for(i=1, T, polinterpolate(v,, x0));
t2 = gettime(); [t1, t2];
}
? p = 101;
? bench(p, 4, 10^4) \\ both methods are equivalent
%3 = [39, 40]
? bench(p, 40) \\ with 40 points formal is much faster
%4 = [45, 355]
```

As the cardinality increases, formal interpolation requires more points to become interesting:

```
? p = nextprime(2^{128});
? bench(p, 4) \setminus formal is slower
\%3 = [16, 9]
? bench(p, 10) \setminus formal has become faster
\%4 = [61, 70]
? bench(p, 100) \\ formal is much faster
\%5 = [1682, 9081]
? p = nextprime(10^{500});
? bench(p, 4) \setminus formal is slower
\%7 = [72, 354]
? bench(p, 20) \\ formal is still slower
\%8 = [1287, 962]
? bench(p, 40) \setminus formal has become faster
\%9 = [3717, 4227]
? bench(p, 100) \\ faster but relatively less impressive
\%10 = [16237, 32335]
```

If t is a complex numeric value and e is present, e will contain an error estimate on the returned value. More precisely, let P be the interpolation polynomial on the given n points; there exist a subset of n - 1 points and Q the attached interpolation interpolation polynomial such that e = exponent(P(t) - Q(t)) (Neville's algorithm).

```
? f(x) = 1 / (1 + 25*x^2);
2 \times 0 = 975/1000;
? test(X) =
 { my (P, e);
P = polinterpolate(X, [f(x) | x <- X], x0, \&e);
[ exponent(P - f(x0)), e ];
}
\\ equidistant nodes vs. Chebyshev nodes
? test( [-10..10] / 10 )
\%4 = [6, 5]
? test( polrootsreal(polchebyshev(21)) )
\%5 = [-15, -10]
? test( [-100..100] / 100 )
\%7 = [93, 97] \setminus P(x0) is way different from f(x0)
? test( polrootsreal(polchebyshev(201)) )
\%8 = [-60, -55]
```

This is an example of Runge's phenomenon: increasing the number of equidistant nodes makes extrapolation much worse. Note that the error estimate is not a guaranteed upper bound (cf %4), but is reasonably tight in practice.

poliscyclo(f)

Returns 0 if f is not a cyclotomic polynomial, and n > 0 if $f = \Phi_n$, the n-th cyclotomic polynomial.

```
? poliscyclo(x^4-x^2+1)
%1 = 12
? polcyclo(12)
%2 = x^4 - x^2 + 1
? poliscyclo(x^4-x^2-1)
%3 = 0
```

poliscycloprod(f)

Returns 1 if f is a product of cyclotomic polynomial, and 0 otherwise.

```
? f = x^6+x^5-x^3+x+1;
? poliscycloprod(f)
%2 = 1
? factor(f)
%3 =
[ x^2 + x + 1 1]
[x^4 - x^2 + 1 1]
? [ poliscyclo(T) | T <- %[,1] ]
%4 = [3, 12]
? polcyclo(3) * polcyclo(12)
%5 = x^6 + x^5 - x^3 + x + 1
```

polisirreducible(pol)

pol being a polynomial (univariate in the present version **2.13.3**), returns 1 if *pol* is nonconstant and irreducible, 0 otherwise. Irreducibility is checked over the smallest base field over which *pol* seems to be defined.

pollaguerre(n, a, b, flag)

n - th Laguerre polynomial $L_n^{(a)}$ of degree n and parameter a evaluated at b ('x by default), i.e.

$$L_n^{(a)}(x) = (x^{-a}e^x)/(n!)(d^n)/(dx^n)(e^{-x}x^{n+a}).$$

If flag is 1, return $[L_{n-1}^{(a)}(b), L_n^{(a)}(b)]$.

pollead(x, v)

Leading coefficient of the polynomial or power series x. This is computed with respect to the main variable of x if v is omitted, with respect to the variable v otherwise.

pollegendre(n, a, flag)

n-th Legendre polynomial P_n evaluated at a ('x by default), where

$$P_n(x) = (1)/(2^n n!)(d^n)/(dx^n)(x^2 - 1)^n.$$

If *flag* is 1, return $[P_{n-1}(a), P_n(a)]$.

polmodular(L, inv, x, y, derivs)

Return the modular polynomial of prime level L in variables x and y for the modular function specified by inv. If inv is 0 (the default), use the modular j function, if inv is 1 use the Weber-f function, and if inv is 5 use $\gamma_2 = \sqrt[3]{j}$. See polclass for the full list of invariants. If x is given as Mod(j, p) or an element j of a finite field (as a t_FFELT), then return the modular polynomial of level L evaluated at j. If j is from a finite field and derivs is nonzero, then return a triple where the last two elements are the first and second derivatives of the modular polynomial evaluated at j.

```
? polmodular(3)
%1 = x^4 + (-y^3 + 2232*y^2 - 1069956*y + 36864000)*x^3 + \dots
? polmodular(7, 1, , 'J)
\%2 = x^8 - J^7 x^7 + 7^J^4 x^4 - 8^J x + J^8
? polmodular(7, 5, 7*ffgen(19)^0, 'j)
\%3 = j^8 + 4*j^7 + 4*j^6 + 8*j^5 + j^4 + 12*j^2 + 18*j + 18
? polmodular(7, 5, Mod(7,19), 'j)
%4 = Mod(1, 19)*j^8 + Mod(4, 19)*j^7 + Mod(4, 19)*j^6 + ...
? u = ffgen(5)^0; T = polmodular(3,0,,'j)*u;
? polmodular(3, 0, u,'j,1)
\%6 = [j^4 + 3^*j^2 + 4^*j + 1, 3^*j^2 + 2^*j + 4, 3^*j^3 + 4^*j^2 + 4^*i + 2]
? subst(T,x,u)
\%7 = j^4 + 3*j^2 + 4*j + 1
? subst(T',x,u)
\%8 = 3*j^2 + 2*j + 4
? subst(T'',x,u)
\%9 = 3*j^3 + 4*j^2 + 4*j + 2
```

polrecip(pol)

Reciprocal polynomial of *pol* with respect to its main variable, i.e. the coefficients of the result are in reverse order; *pol* must be a polynomial.

? polrecip(x² + 2*x + 3) %1 = 3*x² + 2*x + 1 ? polrecip(2*x + y) %2 = y*x + 2

polred(T, flag, _arg3)

This function is *deprecated*, use **polredbest** instead. Finds polynomials with reasonably small coefficients

defining subfields of the number field defined by T. One of the polynomials always defines \mathbb{Q} (hence has degree 1), and another always defines the same number field as T if T is irreducible.

All T accepted by nfinit are also allowed here; in particular, the format [T, listP] is recommended, e.g. with $listP = 10^5$ or a vector containing all ramified primes. Otherwise, the maximal order of $\mathbb{Q}[x]/(T)$ must be computed.

The following binary digits of flag are significant:

1: Possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than primelimit or divide integers stored in the addprimes table. This flag is *deprecated*, the [T, listP] format is more flexible.

2: gives also elements. The result is a two-column matrix, the first column giving primitive elements defining these subfields, the second giving the corresponding minimal polynomials.

```
? M = polred(x^4 + 8, 2)
%1 =
[ 1 x - 1]
[ 1/2*x^2 + 1 x^2 - 2*x + 3]
[-1/2*x^2 + 1 x^2 - 2*x + 3]
[ 1/2*x^2 x^2 + 2]
[ 1/4*x^3 x^4 + 2]
? minpoly(Mod(M[2,1], x^4+8))
%2 = x^2 + 2
```

polredabs(T, flag)

Returns a canonical defining polynomial P for the number field $\mathbb{Q}[X]/(T)$ defined by T, such that the sum of the squares of the modulus of the roots (i.e. the T_2 -norm) is minimal. Different T defining isomorphic number fields will yield the same P. All T accepted by nfinit are also allowed here, e.g. nonmonic polynomials, or pairs [T, listP] specifying that a nonmaximal order may be used. For convenience, any number field structure (*nf*, *bnf*,...) can also be used instead of T.

? polredabs(x² + 16) %1 = x² + 1 ? K = bnfinit(x² + 16); polredabs(K) %2 = x² + 1

Warning 1. Using a t_POL T requires computing and fully factoring the discriminant d_K of the maximal order which may be very hard. You can use the format [T, listP], where listP encodes a list of known coprime divisors of disc(T) (see ??nfbasis), to help the routine, thereby replacing this part of the algorithm by a polynomial time computation But this may only compute a suborder of the maximal order, when the divisors are not squarefree or do not include all primes dividing d_K . The routine attempts to certify the result independently of this order computation as per nfcertify: we try to prove that the computed order is maximal. If the certification fails, the routine then fully factors the integers returned by nfcertify. You can also use polredbest to avoid this factorization step; in this case, the result is small but no longer canonical.

Warning 2. Apart from the factorization of the discriminant of T, this routine runs in polynomial time for a *fixed* degree. But the complexity is exponential in the degree: this routine may be exceedingly slow when the number field has many subfields, hence a lot of elements of small T_2 -norm. If you do not need a canonical polynomial, the function **polredbest** is in general much faster (it runs in polynomial time), and tends to return polynomials with smaller discriminants.

The binary digits of flag mean

1: outputs a two-component row vector [P, a], where P is the default output and Mod(a, P) is a root of the original T.

4: gives all polynomials of minimal T_2 norm; of the two polynomials P(x) and P(-x), only one is given.

16: (OBSOLETE) Possibly use a suborder of the maximal order, *without* attempting to certify the result as in Warning 1. This makes polredbas behave like polredbest. Just use the latter.

```
? T = x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 \
        - 7453176*x^6 + 13950764*x^4 - 5596840*x^2 + 46225
? T1 = polredbs(T); T2 = polredbest(T);
? [ norml2(polroots(T1)), norml2(polroots(T2)) ]
%3 = [88.0000000, 120.000000]
? [ sizedigit(poldisc(T1)), sizedigit(poldisc(T2)) ]
%4 = [75, 67]
```

The precise definition of the output of polredabs is as follows.

- Consider the finite list of characteristic polynomials of primitive elements of K that are in \mathbb{Z}_K and minimal for the T_2 norm; now remove from the list the polynomials whose discriminant do not have minimal absolute value. Note that this condition is restricted to the original list of polynomials with minimal T_2 norm and does not imply that the defining polynomial for the field with smallest discriminant belongs to the list !
- To a polynomial $P(x) = x^n + ... + a_n \in \mathbb{R}[x]$ we attach the sequence S(P) given by $||a_1||, a_1, ..., ||a_n||, a_n$. Order the polynomials P by the lexicographic order on the coefficient vectors S(P). Then the output of polredabs is the smallest polynomial in the above list for that order. In other words, the monic polynomial which is lexicographically smallest with respect to the absolute values of coefficients, favouring negative coefficients to break ties, i.e. choosing $x^3 - 2$ rather than $x^3 + 2$.

polredbest(T, flag)

Finds a polynomial with reasonably small coefficients defining the same number field as *T*. All *T* accepted by nfinit are also allowed here (e.g. nonmonic polynomials, nf, bnf, [T,Z_K_basis]). Contrary to polredabs, this routine runs in polynomial time, but it offers no guarantee as to the minimality of its result.

This routine computes an LLL-reduced basis for an order in $\mathbb{Q}[X]/(T)$, then examines small linear combinations of the basis vectors, computing their characteristic polynomials. It returns the *separable* polynomial P of smallest discriminant, the one with lexicographically smallest abs(Vec(P)) in case of ties. This is a good candidate for subsequent number field computations since it guarantees that the denominators of algebraic integers, when expressed in the power basis, are reasonably small. With no claim of minimality, though.

It can happen that iterating this functions yields better and better polynomials, until it stabilizes:

```
? \p5
? P = X^12+8*X^8-50*X^6+16*X^4-3069*X^2+625;
? poldisc(P)*1.
%2 = 1.2622 E55
? P = polredbest(P);
? poldisc(P)*1.
%4 = 2.9012 E51
? P = polredbest(P);
? poldisc(P)*1.
%6 = 8.8704 E44
```

In this example, the initial polynomial P is the one returned by polredabs, and the last one is stable.

If flag = 1: outputs a two-component row vector [P, a], where P is the default output and Mod(a, P) is a root of the original T.

? [P,a] = polredbest(x⁴ + 8, 1) %1 = [x⁴ + 2, Mod(x³, x⁴ + 2)] ? charpoly(a) %2 = x⁴ + 8

In particular, the map $\mathbb{Q}[x]/(T) \to \mathbb{Q}[x]/(P)$, x : -- > Mod(a, P) defines an isomorphism of number fields, which can be computed as

subst(lift(Q), 'x, a)

if Q is a t_POLMOD modulo T; b = modreverse(a) returns a t_POLMOD giving the inverse of the above map (which should be useless since $\mathbb{Q}[x]/(P)$ is a priori a better representation for the number field and its elements).

polredord(x)

This function is obsolete, use polredbest.

polresultant(x, y, v, flag)

Resultant of the two polynomials x and y with exact entries, with respect to the main variables of x and y if v is omitted, with respect to the variable v otherwise. The algorithm assumes the base ring is a domain. If you also need the u and v such that x * u + y * v = Res(x, y), use the polresultantext function.

If flag = 0 (default), uses the algorithm best suited to the inputs, either the subresultant algorithm (Lazard/Ducos variant, generic case), a modular algorithm (inputs in $\mathbb{Q}[X]$) or Sylvester's matrix (inexact inputs).

If flag = 1, uses the determinant of Sylvester's matrix instead; this should always be slower than the default.

If x or y are multivariate with a huge *polynomial* content, it is advisable to remove it before calling this function. Compare:

? a = polcyclo(7) * ((t+1)/(t+2))^100; ? b = polcyclo(11)* ((t+2)/(t+3))^100); ? polresultant(a,b); time = 3,833 ms. ? ca = content(a); cb = content(b); \ polresultant(a/ca,b/cb)*ca^poldegree(b)*cb*poldegree(a); \\ instantaneous

The function only removes rational denominators and does not compute automatically the content because it is generically small and potentially *very* expensive (e.g. in multivariate contexts). The choice is yours, depending on your application.

polresultantext(A, B, v)

Finds polynomials U and V such that A * U + B * V = R, where R is the resultant of U and V with respect to the main variables of A and B if v is omitted, and with respect to v otherwise. Returns the row vector [U, V, R]. The algorithm used (subresultant) assumes that the base ring is a domain.

? A = x*y; B = (x+y)^2; ? [U,V,R] = polresultantext(A, B) %2 = [-y*x - 2*y^2, y^2, y^4] ? A*U + B*V %3 = y^4 ? [U,V,R] = polresultantext(A, B, y) %4 = [-2*x^2 - y*x, x^2, x^4] ? A*U+B*V %5 = x^4

polroots(T, precision)

Complex roots of the polynomial T, given as a column vector where each root is repeated according to its multiplicity and given as floating point complex numbers at the current realprecision:

? polroots(x^2) %1 = [0.E-38 + 0.E-38*I, 0.E-38 + 0.E-38*I]~ ? polroots(x^3+1) %2 = [-1.00... + 0.E-38*I, 0.50... - 0.866...*I, 0.50... + 0.866...*I]~

The algorithm used is a modification of Schönhage's root-finding algorithm, due to and originally implemented by Gourdon. It runs in polynomial time in deg(T) and the precision. If furthermore T has rational coefficients, roots are guaranteed to the required relative accuracy. If the input polynomial T is exact, then the ordering of the roots does not depend on the precision: they are ordered by increasing $||\Im z||$, then by increasing $\Re z$; in case of tie (conjugates), the root with negative imaginary part comes first.

polrootsbound(T, tau)

Return a sharp upper bound B for the modulus of the largest complex root of the polynomial T with complex coefficients with relative error τ . More precisely, we have $||z|| \le B$ for all roots and there exist one root such that $||z_0|| \ge B \exp(-2\tau)$. Much faster than either polynomials or polynomial.

```
? T=poltchebi(500);
? vecmax(abs(polroots(T)))
time = 5.706 ms.
%2 = 0.99999506520185816611184481744870013191
? vecmax(abs(polrootsreal(T)))
time = 1,972 ms.
%3 = 0.99999506520185816611184481744870013191
? polrootsbound(T)
time = 217 \text{ ms}.
\%4 = 1.0098792554165905155
? polrootsbound(T, log(2)/2) \\ allow a factor 2, much faster
time = 51 \text{ ms}.
\%5 = 1.4065759938190154354
? polrootsbound(T, 1e-4)
time = 504 \text{ ms}.
\%6 = 1.0000920717983847741
? polrootsbound(T, 1e-6)
time = 810 \text{ ms}.
%7 = 0.9999960628901692905
? polrootsbound(T, 1e-10)
time = 1.351 ms.
\%8 = 0.9999950652993869760
```

polrootsff(x, p, a)

Obsolete, kept for backward compatibility: use factormod.

polrootsmod(f, D)

Vector of roots of the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T(y) an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[y]/(T)$ (as usual the main variable of T must have lower priority than the main variable of f);
- *D* a t_FFELT: factor over the attached field;

• D omitted: factor over the field of definition of f, which must be a finite field.

Multiple roots are not repeated.

```
? polrootsmod(x^2-1,2)
%1 = [Mod(1, 2)]~
? polrootsmod(x^2+1,3)
%2 = []~
? polrootsmod(x^2+1, [y^2+1,3])
%3 = [Mod(Mod(1, 3)*y, Mod(1, 3)*y^2 + Mod(1, 3)),
Mod(Mod(2, 3)*y, Mod(1, 3)*y^2 + Mod(1, 3))]~
? polrootsmod(x^2 + Mod(1,3))
%4 = []~
? liftall( polrootsmod(x^2 + Mod(Mod(1,3),y^2+1)) )
%5 = [y, 2*y]~
? t = ffgen(y^2+Mod(1,3)); polrootsmod(x^2 + t^0)
%6 = [y, 2*y]~
```

polrootspadic(f, p, r)

Vector of *p*-adic roots of the polynomial *pol*, given to *p*-adic precision *r*; the integer *p* is assumed to be a prime. Multiple roots are *not* repeated. Note that this is not the same as the roots in $\mathbb{Z}/p^r\mathbb{Z}$, rather it gives approximations in $\mathbb{Z}/p^r\mathbb{Z}$ of the true roots living in \mathbb{Q}_p :

```
? polrootspadic(x^3 - x^2 + 64, 2, 4)
%1 = [2^3 + 0(2^4), 2^3 + 0(2^4), 1 + 0(2^4)]~
? polrootspadic(x^3 - x^2 + 64, 2, 5)
%2 = [2^3 + 0(2^5), 2^3 + 2^4 + 0(2^5), 1 + 0(2^5)]~
```

As the second commands show, the first two roots *are* distinct in \mathbb{Q}_p , even though they are equal modulo 2^4 .

More generally, if T is an integral polynomial irreducible mod p and f has coefficients in $\mathbb{Q}[t]/(T)$, the argument p may be replaced by the vector [T, p]; we then return the roots of f in the unramified extension $\mathbb{Q}_p[t]/(T)$.

```
? polrootspadic(x<sup>3</sup> - x<sup>2</sup> + 64*y, [y<sup>2</sup>+y+1,2], 5)
%3 = [Mod((2<sup>3</sup> + 0(2<sup>5</sup>))*y + (2<sup>3</sup> + 0(2<sup>5</sup>)), y<sup>2</sup> + y + 1),
Mod((2<sup>3</sup> + 2<sup>4</sup> + 0(2<sup>5</sup>))*y + (2<sup>3</sup> + 2<sup>4</sup> + 0(2<sup>5</sup>)), y<sup>2</sup> + y + 1),
Mod(1 + 0(2<sup>5</sup>), y<sup>2</sup> + y + 1)]~
```

If *pol* has inexact t_PADIC coefficients, this need not well-defined; in this case, the polynomial is first made integral by dividing out the *p*-adic content, then lifted to \mathbb{Z} using truncate coefficientwise. Hence the roots given are approximations of the roots of an exact polynomial which is *p*-adically close to the input. To avoid pitfalls, we advise to only factor polynomials with exact rational coefficients.

polrootsreal(T, ab, precision)

Real roots of the polynomial T with real coefficients, multiple roots being included according to their multiplicity. If the polynomial does not have rational coefficients, it is first rescaled and rounded. The roots are given to a relative accuracy of realprecision. If argument ab is present, it must be a vector [a, b] with two components (of type t_INT, t_FRAC or t_INFINITY) and we restrict to roots belonging to that closed interval.

```
? \p9
? polrootsreal(x^2-2)
%1 = [-1.41421356, 1.41421356]~
? polrootsreal(x^2-2, [1,+oo])
%2 = [1.41421356]~
? polrootsreal(x^2-2, [2,3])
```

(continues on next page)

(continued from previous page)

```
%3 = []~
? polrootsreal((x-1)*(x-2), [2,3])
%4 = [2.00000000]~
```

The algorithm used is a modification of Uspensky's method (relying on Descartes's rule of sign), following Rouillier and Zimmerman's article "Efficient isolation of a polynomial real roots" (http://hal.inria.fr/inria-00072518/). Barring bugs, it is guaranteed to converge and to give the roots to the required accuracy.

Remark. If the polynomial T is of the form $Q(x^h)$ for some $h \ge 2$ and ab is omitted, the routine will apply the algorithm to Q (restricting to nonnegative roots when h is even), then take h-th roots. On the other hand, if you want to specify ab, you should apply the routine to Q yourself and a suitable interval [a', b'] using approximate h-th roots adapted to your problem: the function will not perform this change of variables if ab is present.

polsturm(T, ab, _arg3)

Number of distinct real roots of the real polynomial T. If the argument ab is present, it must be a vector [a, b] with two real components (of type t_INT, t_REAL, t_FRAC or t_INFINITY) and we count roots belonging to that closed interval.

If possible, you should stick to exact inputs, that is avoid t_REAL s in T and the bounds a, b: the result is then guaranteed and we use a fast algorithm (Uspensky's method, relying on Descartes's rule of sign, see polrootsreal). Otherwise, the polynomial is rescaled and rounded first and the result may be wrong due to that initial error. If only a or b is inexact, on the other hand, the interval is first thickened using rational endpoints and the result remains guaranteed unless there exist a root *very* close to a nonrational endpoint (which may be missed or unduly included).

```
? T = (x-1)*(x-2)*(x-3);
? polsturm(T)
%2 = 3
? polsturm(T, [-oo,2])
%3 = 2
? polsturm(T, [1/2,+oo])
%4 = 3
? polsturm(T, [1, Pi]) \\ Pi inexact: not recommended !
%5 = 3
? polsturm(T*1., [0, 4]) \\ T*1. inexact: not recommended !
%6 = 3
? polsturm(T*2, [0, 4]) \\ not squarefree: roots are not repeated!
%7 = 3
```

polsubcyclo(n, d, v)

Gives polynomials (in variable v) defining the sub-Abelian extensions of degree d of the cyclotomic field $\mathbb{Q}(\zeta_n)$, where $d \| \phi(n)$.

If there is exactly one such extension the output is a polynomial, else it is a vector of polynomials, possibly empty. To get a vector in all cases, use concat([], polsubcyclo(n,d)).

The function galoissubcyclo allows to specify exactly which sub-Abelian extension should be computed.

polsylvestermatrix(x, y)

Forms the Sylvester matrix corresponding to the two polynomials x and y, where the coefficients of the polynomials are put in the columns of the matrix (which is the natural direction for solving equations afterwards). The use of this matrix can be essential when dealing with polynomials with inexact entries, since polynomial Euclidean division doesn't make much sense in this case.

polsym(x, n)

Creates the column vector of the symmetric powers of the roots of the polynomial x up to power n, using Newton's formula.

poltchebi(n, v)

Deprecated alias for polchebyshev

polteichmuller(T, p, r)

Given $T \in \mathbb{F}_p[X]$ return the polynomial $P \in \mathbb{Z}_p[X]$ whose roots (resp. leading coefficient) are the Teichmuller lifts of the roots (resp. leading coefficient) of T, to p-adic precision r. If T is monic, P is the reduction modulo p^r of the unique monic polynomial congruent to T modulo p such that $P(X^p) = 0(modP(X), p^r)$.

```
? T = ffinit(3, 3, 't)
%1 = Mod(1,3)*t^3 + Mod(1,3)*t^2 + Mod(1,3)*t + Mod(2,3)
? P = polteichmuller(T,3,5)
%2 = t^3 + 166*t^2 + 52*t + 242
? subst(P, t, t^3) % (P*Mod(1,3^5))
%3 = Mod(0, 243)
? [algdep(a+0(3^5),2) | a <- Vec(P)]
%4 = [x - 1, 5*x^2 + 1, x^2 + 4*x + 4, x + 1]
```

When T is monic and irreducible mod p, this provides a model $\mathbb{Q}_p[X]/(P)$ of the unramified extension $\mathbb{Q}_p[X]/(T)$ where the Frobenius has the simple form $X \mod P : --> X^p \mod P$.

poltschirnhaus(x)

Applies a random Tschirnhausen transformation to the polynomial x, which is assumed to be nonconstant and separable, so as to obtain a new equation for the étale algebra defined by x. This is for instance useful when computing resolvents, hence is used by the **polgalois** function.

polylog(m, x, flag, precision)

One of the different polylogarithms, depending on *flag*:

If flag = 0 or is omitted: m - th polylogarithm of x, i.e. analytic continuation of the power series $Li_m(x) = \sum_{n>=1} x^n/n^m$ (x < 1). Uses the functional equation linking the values at x and 1/x to restrict to the case ||x|| <= 1, then the power series when $||x||^2 <= 1/2$, and the power series expansion in $\log(x)$ otherwise.

Using flag, computes a modified m - th polylogarithm of x. We use Zagier's notations; let \Re_m denote \Re or \Im depending on whether m is odd or even:

If flag = 1: compute $D_m(x)$, defined for $||x|| \le 1$ by

$$\Re_m \left(\sum_{k=0}^{m-1} \left(\left(-\log \|x\|\right)^k\right) / (k!) Li_{m-k}(x) + \left(\left(-\log \|x\|\right)^{m-1}\right) / (m!) \log \|1-x\|\right)\right)$$

If flag = 2: compute $D_m(x)$, defined for $||x|| \le 1$ by

$$\Re_m \left(\sum_{k=0}^{m-1} \left((-\log \|x\|)^k\right) / (k!) Li_{m-k}(x) - (1) / (2) \left((-\log \|x\|)^m\right) / (m!)\right).$$

If flag = 3: compute $P_m(x)$, defined for $||x|| \le 1$ by

$$\Re_m(\sum_{k=0}^{m-1} (2^k B_k)/(k!)(\log ||x||)^k Li_{m-k}(x) - (2^{m-1} B_m)/(m!)(\log ||x||)^m).$$

These three functions satisfy the functional equation $f_m(1/x) = (-1)^{m-1} f_m(x)$.

polylogmult(s, z, t, precision)

For s a vector of positive integers and z a vector of complex numbers of the same length, returns the multiple polylogarithm value (MPV)

$$\zeta(s_1, \dots, s_r; z_1, \dots, z_r) = \sum_{n_1 > \dots > n_r > 0} \prod_{1 < \dots > n_r > 0} \sum_{1 < i < r} z_i^{n_i} / n_i^{s_i}.$$

If z is omitted, assume z = [1, ..., 1], i.e., Multiple Zeta Value. More generally, return Yamamoto's interpolation between ordinary multiple polylogarithms (t = 0) and star polylogarithms (t = 1, using the condition $n_1 > =$... $> = n_r > 0$), evaluated at t.

We must have $||z_1...z_i|| \le 1$ for all *i*, and if $s_1 = 1$ we must have $z_1! = 1$.

? 8*polylogmult([2,1],[-1,1]) - zeta(3) %1 = 0.E-38

Warning. The algorithm used converges when the z_i are 1. It may not converge as some $z_i! = 1$ becomes too close to 1, even at roots of 1 of moderate order:

```
? polylogmult([2,1], (99+20*I)/101 * [1,1])
   *** polylogmult: sorry, polylogmult in this range is not yet implemented.
? polylogmult([2,1], exp(I*Pi/20)* [1,1])
   *** polylogmult: sorry, polylogmult in this range is not yet implemented.
```

More precisely, if $y_i := 1/(z_1...z_i)$ and

$$v := \min_{i < j; y_i! = 1} \| (1 - y_i) y_j \| > 1/4$$

then the algorithm computes the value up to a 2^{-b} absolute error in $O(k^2N)$ operations on floating point numbers of O(N) bits, where $k = \sum_i s_i$ is the weight and $N = b/\log_2(4v)$.

polzagier(n, m)

Creates Zagier's polynomial $P_n^{(m)}$ used in the functions sumalt and sumpos (with flag = 1), see "Convergence acceleration of alternating series", Cohen et al., *Experiment. Math.*, vol. 9, 2000, pp. 3–12.

If m < 0 or m >= n, $P_n^{(m)} = 0$. We have $P_n := P_n^{(0)}$ is $T_n(2x - 1)$, where T_n is the Legendre polynomial of the second kind. For n > m > 0, $P_n^{(m)}$ is the *m*-th difference with step 2 of the sequence $n^{m+1}P_n$; in this case, it satisfies

$$2P_n^{(m)}(\sin^2 t) = (d^{m+1})/(dt^{m+1})(\sin(2t)^m \sin(2(n-m)t)).$$

powers(x, n, x0)

For nonnegative n, return the vector with n+1 components $[1, x, ..., x^n]$ if **x0** is omitted, and $[x_0, x_0 * x, ..., x_0 * x^n]$ otherwise.

? powers(Mod(3,17), 4) %1 = [Mod(1, 17), Mod(3, 17), Mod(9, 17), Mod(10, 17), Mod(13, 17)] ? powers(Mat([1,2;3,4]), 3) %2 = [[1, 0; 0, 1], [1, 2; 3, 4], [7, 10; 15, 22], [37, 54; 81, 118]] ? powers(3, 5, 2) %3 = [2, 6, 18, 54, 162, 486]

When n < 0, the function returns the empty vector [].

precision(x, n)

The function behaves differently according to whether n is present or not. If n is missing, the function returns the floating point precision in decimal digits of the PARI object x. If x has no floating point component, the function returns +00.

```
? precision(exp(1e-100))
%1 = 154 \\ 154 significant decimal digits
? precision(2 + x)
%2 = +oo \\ exact object
? precision(0.5 + 0(x))
%3 = 38 \\ floating point accuracy, NOT series precision
? precision( [ exp(1e-100), 0.5 ] )
%4 = 38 \\ minimal accuracy among components
```

Using getlocalprec() allows to retrieve the working precision (as modified by possible localprec statements).

If *n* is present, the function creates a new object equal to *x* with a new floating point precision *n*: *n* is the number of desired significant *decimal* digits. If *n* is smaller than the precision of a t_REAL component of *x*, it is truncated, otherwise it is extended with zeros. For non-floating-point types, no change.

precprime(x)

Finds the largest pseudoprime (see ispseudoprime) less than or equal to x. x can be of any real type. Returns 0 if $x \le 1$. Note that if x is a prime, this function returns x and not the largest prime strictly smaller than x. To rigorously prove that the result is prime, use isprime.

prime(n)

The n - th prime number

? prime(10^9)
%1 = 22801763489

Uses checkpointing and a naive O(n) algorithm. Will need about 30 minutes for n up to 10^{11} ; make sure to start gp with primelimit at least $\sqrt{p_n}$, e.g. the value $\sqrt{n \log(n \log n)}$ is guaranteed to be sufficient.

primecert(N, flag)

If N is a prime, return a PARI Primality Certificate for the prime N, as described below. Otherwise, return 0. A Primality Certificate c can be checked using primecertisvalid(c).

If flag = 0 (default), return an ECPP certificate (Atkin-Morain)

A PARI ECPP Primality Certificate for the prime N is either a prime integer $N < 2^{64}$ or a vector C of length ℓ whose i is a vector $[N_i, t_i, s_i, a_i, P_i]$ of length 5 where $N_1 = N$. It is said to be *valid* if for each $i = 1, ..., \ell$, all of the following conditions are satisfied

- N_i is a positive integer
- t_i is an integer such that $t_i^2 < 4N_i$
- s_i is a positive integer which divides m_i where $m_i = N_i + 1 t_i$
- If we set $q_i = (m_i)/(s_i)$, then

*
$$q_i > (N_i^{1/4} + 1)^2$$

* $q_i = N_{i+1}$ if $1 \le i < l$

* $q_{\ell} <= 2^{64}$ is prime

• a_i is an integer

* P[i] is a vector of length 2 representing the affine point $P_i = (x_i, y_i)$ on the elliptic curve $E : y^2 = x^3 + a_i x + b_i$ modulo N_i where $b_i = y_i^2 - x_i^3 - a_i x_i$ satisfying the following:

* $m_i P_i = oo$

 $* s_i P_i! = oo$

Using the following theorem, the data in the vector C allows to recursively certify the primality of N (and all the q_i) under the single assumption that q_ℓ be prime.

Theorem. If N is an integer and there exist positive integers m, q and a point P on the elliptic curve E: $y^2 = x^3 + ax + b$ defined modulo N such that $q > (N^{1/4} + 1)^2$, q is a prime divisor of m, mP = oo and (m)/(q)P! = oo, then N is prime.

If flag = 1 (very slow), return an N - 1 certificate (Pocklington Lehmer)

A PARI N - 1 Primality Certificate for the prime N is either a prime integer $N < 2^{64}$ or a pair [N, C], where C is a vector with ℓ elements which are either a single integer $p_i < 2^{64}$ or a triple $[p_i, a_i, C_i]$ with $p_i > 2^{64}$ satisfying the following properties:

- p_i is a prime divisor of N-1;
- a_i is an integer such that $a_i^{N-1} = 1 \pmod{N}$ and $a_i^{(N-1)/p_i} 1$ is coprime with N;
- C_i is an N-1 Primality Certificate for p_i
- The product F of the $p_i^{v_{p_i}(N-1)}$ is strictly larger than $N^{1/3}$. Provided that all p_i are indeed primes, this implies that any divisor of N is congruent to 1 modulo F.
- The Billhart, Lehmer, Selfridge criterion is satisfied: when we write $N = 1 + c_1F + c_2F^2$ in base F the polynomial $1 + c_1X + c_2X^2$ is irreducible over \mathbb{Z} , i.e. $c_1^2 4c_2$ is not a square. This implies that N is prime.

This algorithm requires factoring partially p - 1 for various prime integers p with an unfactored parted $\langle = p^{2/3}$ and this may be exceedingly slow compared to the default.

The algorithm fails if one of the pseudo-prime factors is not prime, which is exceedingly unlikely and well worth a bug report. Note that if you monitor the algorithm at a high enough debug level, you may see warnings about untested integers being declared primes. This is normal: we ask for partial factorizations (sufficient to prove primality if the unfactored part is not too large), and factor warns us that the cofactor hasn't been tested. It may or may not be tested later, and may or may not be prime. This does not affect the validity of the whole Primality Certificate.

primecertexport(cert, format)

Returns a string suitable for print/write to display a primality certificate from primecert, the format of which depends on the value of format:

- 0 (default): Human-readable format. See ??primecert for the meaning of the successive N, t, s, a, m, q, E, P. The integer D is the negative fundamental discriminant coredisc $(t^2 4N)$.
- 1: Primo format 4.
- 2: MAGMA format.

Currently, only ECPP Primality Certificates are supported.

```
? cert = primecert(10^35+69);
? s = primecertexport(cert); \\ Human-readable
? print(s)
[1]
t = 546867911035452074
s = 2963504668391148
a = 0
D = -3
m = 999999999999999999453132088964547996
q = 33743830764501150277
E = [0, 1]
P = [21567861682493263464353543707814204],
49167839501923147849639425291163552]
۲<u>2</u>1
N = 33743830764501150277
t = -11610830419
s = 734208843
a = 0
D = -3
m = 33743830776111980697
q = 45959444779
E = [0, 25895956964997806805]
P = [29257172487394218479, 3678591960085668324]
\\ Primo format
? s = primecertexport(cert,1); write("cert.out", s);
\\ Magma format, write to file
? s = primecertexport(cert,2); write("cert.m", s);
? cert = primecert(10^{35+69}, 1); \\ N-1 certificate
? primecertexport(cert)
 *** at top-level: primecertexport(cert)
        _____
 *** primecertexport: sorry, N-1 certificate is not yet implemented.
```

primecertisvalid(cert)

Verifies if cert is a valid PARI ECPP Primality certificate, as described in ??primecert.

(continues on next page)

(continued from previous page)

```
, [18022351516, 9326882 51]]]
? primecertisvalid(cert)
%2 = 1
? cert[1][1]++; \\ random perturbation
? primecertisvalid(cert)
%4 = 0 \\ no longer valid
? primecertisvalid(primecert(6))
%5 = 0
```

primepi(x)

The prime counting function. Returns the number of primes $p, p \le x$.

```
? primepi(10)
%1 = 4;
? primes(5)
%2 = [2, 3, 5, 7, 11]
? primepi(10^11)
%3 = 4118054813
```

Uses checkpointing and a naive O(x) algorithm; make sure to start gp with primelimit at least \sqrt{x} .

primes(n)

Creates a row vector whose components are the first n prime numbers. (Returns the empty vector for $n \le 0$.) A t_VEC n = [a, b] is also allowed, in which case the primes in [a, b] are returned

```
? primes(10) \\ the first 10 primes
%1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([0,29]) \\ the primes up to 29
%2 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([15,30])
%3 = [17, 19, 23, 29]
```

prodeulerrat(F, s, a, precision)

 $\prod_{p>=a} F(p^s)$, where the product is taken over prime numbers and F is a rational function.

```
? prodeulerrat(1+1/q^3,1)
%1 = 1.1815649490102569125693997341604542605
? zeta(3)/zeta(6)
%2 = 1.1815649490102569125693997341604542606
```

prodnumrat(F, a, precision)

 $\prod_{n>=a} F(n)$, where F-1 is a rational function of degree less than or equal to -2.

```
? prodnumrat(1+1/x^2,1)
%1 = 3.6760779103749777206956974920282606665
```

psdraw(list, flag)

This function is obsolete, use plotexport and write the result to file.

psi(*x*, *precision*)

The ψ -function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

psplothraw(listx, listy, flag)

This function is obsolete, use plothrawexport and write the result to file.

qfauto(G, fl)

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the automorphism group of the associate lattice. Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension grows. G can also be given by an qfisominit structure. See qfisominit for the meaning of fl.

The output is a two-components vector [o, g] where o is the group order and g is the list of generators (as a vector). For each generator H, the equality $G = {}^{t}HGH$ holds.

The interface of this function is experimental and will likely change in the future.

This function implements an algorithm of Plesken and Souvignier, following Souvignier's implementation.

qfautoexport(qfa, flag)

qfa being an automorphism group as output by qfauto, export the underlying matrix group as a string suitable for (no flags or flag = 0) GAP or (flag = 1) Magma. The following example computes the size of the matrix group using GAP:

```
? G = qfauto([2,1;1,2])
%1 = [12, [[-1, 0; 0, -1], [0, -1; 1, 1], [1, 1; 0, -1]]]
? s = qfautoexport(G)
%2 = "Group([[-1, 0], [0, -1]], [[0, -1], [1, 1]], [[1, 1], [0, -1]])"
? extern("echo \"Order("s");\" | gap -q")
%3 = 12
```

qfbclassno(D, flag)

Ordinary class number of the quadratic order of discriminant D, for "small" values of D.

- if D > 0 or flag = 1, use a $O(||D||^{1/2})$ algorithm (compute $L(1, \chi_D)$ with the approximate functional equation). This is slower than quadclassunit as soon as $||D|| 10^2$ or so and is not meant to be used for large D.
- if D < 0 and flag = 0 (or omitted), use a $O(||D||^{1/4})$ algorithm (Shanks's baby-step/giant-step method). It should be faster than quadclassunit for small values of D, say $||D|| < 10^{18}$.

Important warning. In the latter case, this function only implements part of Shanks's method (which allows to speed it up considerably). It gives unconditionnally correct results for $||D|| < 2.10^{10}$, but may give incorrect results for larger values if the class group has many cyclic factors. We thus recommend to double-check results using the function quadclassunit, which is about 2 to 3 times slower in the range $||D|| \in [10^{10}, 10^{18}]$, assuming GRH. We currently have no counter-examples but they should exist: we would appreciate a bug report if you find one.

Warning. Contrary to what its name implies, this routine does not compute the number of classes of binary primitive forms of discriminant D, which is equal to the *narrow* class number. The two notions are the same when D < 0 or the fundamental unit ε has negative norm; when D > 0 and $N\varepsilon > 0$, the number of classes of forms is twice the ordinary class number. This is a problem which we cannot fix for backward compatibility reasons. Use the following routine if you are only interested in the number of classes of forms:

Here are a few examples:

```
? qfbclassno(400000028) \\ D > 0: slow
time = 3,140 ms.
%1 = 1
? quadclassunit(400000028).no
time = 20 ms. \\{ much faster, assume GRH}
%2 = 1
? qfbclassno(-400000028) \\ D < 0: fast enough
time = 0 ms.
%3 = 7253
? quadclassunit(-400000028).no
time = 0 ms.
%4 = 7253
```

See also qfbhclassno.

qfbcompraw(*x*, *y*)

composition of the binary quadratic forms x and y, without reduction of the result. This is useful e.g. to compute a generating element of an ideal. The result is undefined if x and y do not have the same discriminant.

qfbhclassno(x)

Hurwitz class number of x, when x is nonnegative and congruent to 0 or 3 modulo 4, and 0 for other values. For $x > 5.10^5$, we assume the GRH, and use quadclassunit with default parameters.

```
? qfbhclassno(1) \\ not 0 or 3 mod 4
%1 = 0
? qfbhclassno(3)
%2 = 1/3
? qfbhclassno(4)
%3 = 1/2
? qfbhclassno(23)
%4 = 3
```

qfbil(*x*, *y*, *q*)

This function is obsolete, use qfeval.

qfbnucomp(x, y, L)

composition of the primitive positive definite binary quadratic forms x and y (type t_QFI) using the NUCOMP and NUDUPL algorithms of Shanks, à la Atkin. L is any positive constant, but for optimal speed, one should take $L = ||D/4||^{1/4}$, i.e. sqrtnint(abs(D) >> 2,4), where D is the common discriminant of x and y. When x and y do not have the same discriminant, the result is undefined.

The current implementation is slower than the generic routine for small D, and becomes faster when D has about 45 bits.

qfbnupow(x, n, L)

n-th power of the primitive positive definite binary quadratic form x using Shanks's NUCOMP and NUDUPL algorithms; if set, L should be equal to sqrtnint(abs(D) >> 2,4), where D < 0 is the discriminant of x.

The current implementation is slower than the generic routine for small discriminant D, and becomes faster for $D 2^{45}$.

qfbpowraw(x, n)

n-th power of the binary quadratic form x, computed without doing any reduction (i.e. using qfbcompraw). Here n must be nonnegative and $n < 2^{31}$.

qfbprimeform(*x*, *p*, *precision*)

Prime binary quadratic form of discriminant x whose first coefficient is p, where ||p|| is a prime number. By abuse of notation, p = 1 is also valid and returns the unit form. Returns an error if x is not a quadratic residue mod p, or if x < 0 and p < 0. (Negative definite t_QFI are not implemented.) In the case where x > 0, the "distance" component of the form is set equal to zero according to the current precision.

qfbred(*x*, *flag*, *d*, *isd*, *sd*)

Reduces the binary quadratic form x (updating Shanks's distance function if x is indefinite). The binary digits of flag are toggles meaning

1: perform a single reduction step

2: don't update Shanks's distance

The arguments d, *isd*, *sd*, if present, supply the values of the discriminant, $floor\sqrt{d}$, and \sqrt{d} respectively (no checking is done of these facts). If d < 0 these values are useless, and all references to Shanks's distance are irrelevant.

qfbredsl2(x, data)

Reduction of the (real or imaginary) binary quadratic form x, return [y, g] where y is reduced and g in $SL(2, \mathbb{Z})$ is such that g.x = y; *data*, if present, must be equal to [D, sqrtint(D)], where D > 0 is the discriminant of x. In case x is a t_QFR, the distance component is unaffected.

qfbsolve(Q, n, flag)

Solve the equation Q(x, y) = n in coprime integers x and y (primitive solutions), where Q is a binary quadratic form and n an integer, up to the action of the special orthogonal group $G = SO(Q, \mathbb{Z})$, which is isomorphic to the group of units of positive norm of the quadratic order of discriminant D = discQ. If D > 0, G is infinite. If D < -4, G is of order 2, if D = -3, G is of order 6 and if D = -4, G is of order 4.

Binary digits of flag mean: 1: return all solutions if set, else a single solution; return [] if a single solution is wanted (bit unset) but none exist. 2: also include imprimitive solutions.

When flag = 2 (return a single solution, possibly imprimitive), the algorithm returns a solution with minimal content; in particular, a primitive solution exists if and only if one is returned.

The integer *n* can be given by its factorization matrix :emphasis: fa = factor(n) or by the pair [n, fa].

```
? qfbsolve(Qfb(1,0,2), 603) \\ a single primitive solution
%1 = [5, 17]
? qfbsolve(Qfb(1,0,2), 603, 1) \\ all primitive solutions
%2 = [[5, 17], [-19, -11], [19, -11], [5, -17]]
? qfbsolve(Qfb(1,0,2), 603, 2) \\ a single, possibly imprimitive solution
%3 = [5, 17] \\ actually primitive
? qfbsolve(Qfb(1,0,2), 603, 3) \\ all solutions
%4 = [[5, 17], [-19, -11], [19, -11], [5, -17], [-21, 9], [-21, -9]]
? N = 2^128+1; F = factor(N);
? qfbsolve(Qfb(1,0,1), [N,F],1)
%3 = [[-16382350221535464479,8479443857936402504],
[18446744073709551616,-1], [-18446744073709551616,-1],
[16382350221535464479,8479443857936402504]]
```

For fixed Q, assuming the factorisation of n is given, the algorithm runs in probabilistic polynomial time in $\log p$, where p is the largest prime divisor of n, through the computation of square roots of D modulo 4p). The dependency on Q is more complicated: polynomial time in $\log |D||$ if Q is imaginary, but exponential time if Q

is real (through the computation of a full cycle of reduced forms). In the latter case, note that bnfisprincipal provides a solution in heuristic subexponential time assuming the GRH.

qfeval(q, x, y)

Evaluate the quadratic form q (given by a symmetric matrix) at the vector x; if y is present, evaluate the polar form at (x, y); if q omitted, use the standard Euclidean scalar product, corresponding to the identity matrix.

Roughly equivalent to $x \sim * q * y$, but a little faster and more convenient (does not distinguish between column and row vectors):

```
? x = [1,2,3]~; y = [-1,3,1]~; q = [1,2,3;2,2,-1;3,-1,9];
? qfeval(q,x,y)
%2 = 23
? for(i=1,10^6, qfeval(q,x,y))
time = 661ms
? for(i=1,10^6, x~*q*y)
time = 697ms
```

The speedup is noticeable for the quadratic form, compared to $x \sim * q * x$, since we save almost half the operations:

? for(i=1,10^6, qfeval(q,x))
time = 487ms

The special case q = Id is handled faster if we omit q altogether:

```
? qfeval(,x,y)
%6 = 8
? q = matid(#x);
? for(i=1,10^6, qfeval(q,x,y))
time = 529 ms.
? for(i=1,10^6, qfeval(,x,y))
time = 228 ms.
? for(i=1,10^6, x~*y)
time = 274 ms.
```

We also allow t_MAT s of compatible dimensions for x, and return $\mathbf{x} \sim \mathbf{x} \mathbf{q} \mathbf{x}$ in this case as well:

```
? M = [1,2,3;4,5,6;7,8,9]; qfeval(,M) \\ Gram matrix
%5 =
[66 78 90]
[78 93 108]
[90 108 126]
? q = [1,2,3;2,2,-1;3,-1,9];
? for(i=1,10^6, qfeval(q,M))
time = 2,008 ms.
? for(i=1,10^6, M~*q*M)
time = 2,368 ms.
? for(i=1,10^6, qfeval(,M))
time = 1,053 ms.
```

(continues on next page)

(continued from previous page)

? for(i=1,10^6, M~*M) time = 1,171 ms.

If q is a t_QFI or t_QFR, it is implicitly converted to the attached symmetric t_MAT. This is done more efficiently than by direct conversion, since we avoid introducing a denominator 2 and rational arithmetic:

```
? q = Qfb(2,3,4); x = [2,3];
? qfeval(q, x)
%2 = 62
? Q = Mat(q)
%3 =
  [ 2 3/2]
[3/2 4]
? qfeval(Q, x)
%4 = 62
? for (i=1, 10^6, qfeval(q,x))
time = 758 ms.
? for (i=1, 10^6, qfeval(Q,x))
time = 1,110 ms.
```

Finally, when x is a t_MAT with *integral* coefficients, we allow a t_QFI or t_QFR for q and return the binary quadratic form qoM. Again, the conversion to t_MAT is less efficient in this case:

```
? q = Qfb(2,3,4); Q = Mat(q); x = [1,2;3,4];
? qfeval(q, x)
%2 = Qfb(47, 134, 96)
? qfeval(Q,x)
%3 =
[47 67]
[67 96]
? for (i=1, 10^6, qfeval(q,x))
time = 701 ms.
? for (i=1, 10^6, qfeval(Q,x))
time = 1,639 ms.
```

qfgaussred(q)

decomposition into squares of the quadratic form represented by the symmetric matrix q. The result is a matrix whose diagonal entries are the coefficients of the squares, and the off-diagonal entries on each line represent the bilinear forms. More precisely, if (a_{ij}) denotes the output, one has

$$q(x) = \sum_{i} a_{ii} (x_i + \sum_{j!=i} a_{ij} x_j)^2$$

? qfgaussred([0,1;1,0]) %1 = [1/2 1] [-1 -1/2]

This means that $2xy = (1/2)(x + y)^2 - (1/2)(x - y)^2$. Singular matrices are supported, in which case some diagonal coefficients will vanish:

? qfgaussred([1,1;1,1]) %1 = [1 1]

[1 0]

This means that $x^2 + 2xy + y^2 = (x + y)^2$.

qfisom(G, H, fl, grp)

G, H being square and symmetric matrices with integer entries representing positive definite quadratic forms, return an invertible matrix S such that $G = {}^{t}SHS$. This defines a isomorphism between the corresponding lattices. Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension grows. See qfisominit for the meaning of *fl*. If *grp* is given it must be the automorphism group of H. It will be used to speed up the computation.

G can also be given by an qfisominit structure which is preferable if several forms H need to be compared to G.

This function implements an algorithm of Plesken and Souvignier, following Souvignier's implementation.

qfisominit(G, fl, m)

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, return an **isom** structure allowing to compute isomorphisms between G and other quadratic forms faster.

The interface of this function is experimental and will likely change in future release.

If present, the optional parameter fl must be a t_VEC with two components. It allows to specify the invariants used, which can make the computation faster or slower. The components are

- f1[1] Depth of scalar product combination to use.
- f1[2] Maximum level of Bacher polynomials to use.

If present, m must be the set of vectors of norm up to the maximal of the diagonal entry of G, either as a matrix or as given by qfminim. Otherwise this function computes the minimal vectors so it become very lengthy as the dimension of G grows.

qfjacobi(A, precision)

Apply Jacobi's eigenvalue algorithm to the real symmetric matrix A. This returns [L, V], where

- L is the vector of (real) eigenvalues of A, sorted in increasing order,
- V is the corresponding orthogonal matrix of eigenvectors of A.

(continues on next page)

(continued from previous page)

```
? norml2( (A-L[1])*H[,1] ) \\ approximate eigenvector
%5 = 9.403954806578300064 E-38
? norml2(H*H~ - 1)
%6 = 2.350988701644575016 E-38 \\ close to orthogonal
```

qflll(x, flag)

LLL algorithm applied to the *columns* of the matrix x. The columns of x may be linearly dependent. The result is by default a unimodular transformation matrix T such that x.T is an LLL-reduced basis of the lattice generated by the column vectors of x. Note that if x is not of maximal rank T will not be square. The LLL parameters are (0.51, 0.99), meaning that the Gram-Schmidt coefficients for the final basis satisfy $\|\mu_{i,j}\| \le 0.51$, and the Lovász's constant is 0.99.

If flag = 0 (default), assume that x has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in flag = 1.

If flag = 1, assume that x is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (fplll-1.3).

If flag = 2, x should be an integer matrix whose columns are linearly independent. Returns a partially reduced basis for x, using an unpublished algorithm by Peter Montgomery: a basis is said to be *partially reduced* if $||v_iv_j|| \ge ||v_i||$ for any two distinct basis vectors v_i, v_j . This is faster than flag = 1, esp. when one row is huge compared to the other rows (knapsack-style), and should quickly produce relatively short vectors. The resulting basis is *not* LLL-reduced in general. If LLL reduction is eventually desired, avoid this partial reduction: applying LLL to the partially reduced matrix is significantly *slower* than starting from a knapsack-type lattice.

If flag = 3, as flag = 1, but the reduction is performed in place: the routine returns x.T. This is usually faster for knapsack-type lattices.

If flag = 4, as flag = 1, returning a vector [K, T] of matrices: the columns of K represent a basis of the integer kernel of x (not LLL-reduced in general) and T is the transformation matrix such that x.T is an LLL-reduced \mathbb{Z} -basis of the image of the matrix x.

If flag = 5, case as case 4, but x may have polynomial coefficients.

If flag = 8, same as case 0, but x may have polynomial coefficients.

```
? \p500
realprecision = 500 significant digits
? a = 2*cos(2*Pi/97);
? C = 10^{450};
? v = powers(a,48); b = round(matconcat([matid(48),C*v]~));
? p = b * qflll(b)[,1]; \\ tiny linear combination of powers of 'a'
time = 4,470 ms.
? exponent(v * p / C)
\%5 = -1418
? p3 = qflll(b,3)[,1]; \setminus compute in place, faster
time = 3,790 ms.
? p3 == p \ same result
%7 = 1
? p2 = b * qflll(b,2)[,1]; \\ partial reduction: faster, not as good
time = 343 \text{ ms}.
? exponent(v * p2 / C)
\%9 = -1190
```
qflllgram(G, flag)

Same as qflll, except that the matrix G = x * x is the Gram matrix of some lattice vectors x, and not the coordinates of the vectors themselves. In particular, G must now be a square symmetric real matrix, corresponding to a positive quadratic form (not necessarily definite: x needs not have maximal rank). The result is a unimodular transformation matrix T such that x.T is an LLL-reduced basis of the lattice generated by the column vectors of x. See qflll for further details about the LLL implementation.

If flag = 0 (default), assume that G has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in flag = 1.

If flag = 1, assume that G is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (fplll-1.3).

flag = 4: G has integer entries, gives the kernel and reduced image of x.

flag = 5: same as 4, but G may have polynomial coefficients.

qfminim(*x*, *B*, *m*, *flag*, *precision*)

x being a square and symmetric matrix of dimension d representing a positive definite quadratic form, this function deals with the vectors of x whose norm is less than or equal to B, enumerated using the Fincke-Pohst algorithm, storing at most m pairs of vectors: only one vector is given for each pair v. There is no limit if m is omitted: beware that this may be a huge vector! The vectors are returned in no particular order.

The function searches for the minimal nonzero vectors if B is omitted. The behavior is undefined if x is not positive definite (a "precision too low" error is most likely, although more precise error messages are possible). The precise behavior depends on flag.

- If flag = 0 (default), return [N, M, V], where N is the number of vectors enumerated (an even number, possibly larger than 2m), $M \ll B$ is the maximum norm found, and V is a matrix whose columns are found vectors.
- If flag = 1, ignore m and return [M, v], where v is a nonzero vector of length $M \le B$. If no nonzero vector has length $\le B$, return []. If no explicit B is provided, return a vector of smallish norm, namely the vector of smallest length (usually the first one but not always) in an LLL-reduced basis for x.

In these two cases, x must have integral *small* entries: more precisely, we definitely must have $d \cdot ||x||_o o^2 < 2^{53}$ but even that may not be enough. The implementation uses low precision floating point computations for maximal speed and gives incorrect results when x has large entries. That condition is checked in the code and the routine raises an error if large rounding errors occur. A more robust, but much slower, implementation is chosen if the following flag is used:

• If flag = 2, x can have non integral real entries, but this is also useful when x has large integral entries. Return [N, M, V] as in case flag = 0, where M is returned as a floating point number. If x is inexact and B is omitted, the "minimal" vectors in V only have approximately the same norm (up to the internal working accuracy). This version is very robust but still offers no hard and fast guarantee about the result: it involves floating point operations performed at a high floating point precision depending on your input, but done without rigorous tracking of roundoff errors (as would be provided by interval arithmetic for instance). No example is known where the input is exact but the function returns a wrong result.

```
? x = matid(2);
? qfminim(x) \setminus 4 minimal vectors of norm 1: \pm[0,1], \pm[1,0]
%2 = [4, 1, [0, 1; 1, 0]]
? { x = \setminus The Leech lattice
[4, 2, 0, 0, 0, -2, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 0, 0, -2;
2, 4, -2, -2, 0, -2, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 1, -1, -1;
0, -2, 4, 0, -2, 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 0, 0, 1, -1, -1, 0, 0;
0, -2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 1, -1, 1, 0;
```

0, 0, -2, 0, 4, 0, 0, 0, 1, -1, 0, 0, 1, 0, 0, 0, -2, 0, 0, -1, 1, 1, 0, 0; -2, -2, 0, 0, 0, 4, -2, 0, -1, 0,0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, -1, 1, 1; 0, 0, 0, 0, 0, -2, 4, -2, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, 1, -1, 0; 0, 0, 0, 0, 0, 0, -2, 4, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, -1, -1, 0, 1, 0;0, 0, 0, 0, 1,-1, 0, 0, 4, 0,-2, 0, 1, 1, 0,-1, 0, 1, 0, 0, 0, 0, 0, 0, 0: 0, 0, 0, 0, -1, 0, 0, 0, 0, 4, 0, 0, 1, 1, -1, 1, 0, 0, 0, 1, 0, 0, 1, 0: 0, 0, 0, 0, 0, 0, 0, 0, -2, 0, 4, -2, 0, -1, 0, 0, 0, -1, 0, -1, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2, 4, -1, 1, 0, 0, -1, 1, 0, 1, 1, 1, -1, 0;1, 0, -1, 1, 1, 0, 0, -1, 1, 1, 0, -1, 4, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, -1;-1, -1, 1, -1, 0, 0, 1, 0, 1, 1, -1, 1, 0, 4, 1, 1, 0, 0, 1, 1, 0, 1, 0,1; 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 1, 4, 0, 0, 0, 1, 0, 0, 0, 0, 0; 0, 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 1, 0, 4, 0, 0, 0, 0, 1, 1, 0, 0; 0, 0, 1, 0, -2, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 4, 1, 1, 1, 0, 0, 1, 1; 1, 0, 0, 1, 0, 0, -1, 0, 1, 0, -1, 1, 1, 0, 0, 0, 1, 4, 0, 1, 1, 0, 1, 0; 0, 0, 0, -1, 0, 1, 0, -1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 4, 0, 1, 1, 0, 1;-1, -1, 1, 0, -1, 1, 0, -1, 0, 1, -1, 1, 0, 1, 0, 0, 1, 1, 0, 4, 0, 0, 1, 1;0, 0, -1, 1, 1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 4, 1, 0, 1;0, 1, -1, -1, 1, -1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 4, 0, 1; 0,-1, 0, 1, 0, 1,-1, 1, 0, 1, 0,-1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 4, 1; $-2, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 4]; \}$? $qfminim(x,,0) \setminus 0$: don't store minimal vectors time = 121 ms.%4 = [196560, 4, [;]] \\ 196560 minimal vectors of norm 4 ? qfminim(x) \\ store all minimal vectors ! time = 821 ms.? qfminim(x, 0, 2); \\ safe algorithm. Slower and unnecessary here. time = 5.540 ms.%6 = [196560, 4.000061035156250000, [;]]? qfminim(x,,,2); \\ safe algorithm; store all minimal vectors time = 6,602 ms.

In this example, storing 0 vectors limits memory use; storing all of them requires a parisize about 50MB. All minimal vectors are nevertheless enumerated in both cases of course, which means the speedup is likely to be marginal.

qfnorm(x, q)

This function is obsolete, use qfeval.

qforbits(G, V)

Return the orbits of V under the action of the group of linear transformation generated by the set G. It is assumed that G contains minus identity, and only one vector in v, -v should be given. If G does not stabilize V, the function return 0.

In the example below, we compute representatives and lengths of the orbits of the vectors of norm ≤ 3 under the automorphisms of the lattice \mathbb{Z}^6 .

```
? Q=matid(6); G=qfauto(Q); V=qfminim(Q,3);
? apply(x->[x[1],#x],qforbits(G,V))
%2 = [[[0,0,0,0,0,0,1]~,6],[[0,0,0,0,1,-1]~,30],[[0,0,0,1,-1,-1]~,80]]
```

qfparam(G, sol, flag)

Coefficients of binary quadratic forms that parametrize the solutions of the ternary quadratic form G, using the particular solution *sol. flag* is optional and can be 1, 2, or 3, in which case the *flag*-th form is reduced. The default is *flag* = 0 (no reduction).

? G = [1,0,0;0,1,0;0,0,-34]; ? M = qfparam(G, qfsolve(G)) %2 = [3 -10 -3] [-5 -6 5] [1 0 1]

Indeed, the solutions can be parametrized as

$$(3x^{2} - 10xy - 3y^{2})^{2} + (-5x^{2} - 6xy + 5y^{2})^{2} - 34(x^{2} + y^{2})^{2} = 0.$$

? $v = y^2 * M^{1}[1, x/y, (x/y)^{2}]$ ~ %3 = $[3^{x}x^{2} - 10^{y}x - 3^{y}y^{2}, -5^{x}x^{2} - 6^{y}x + 5^{y}y^{2}, -x^{2} - y^{2}]$ ~ ? $v^{G^{x}v}$ %4 = 0

qfperfection(G)

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the perfection rank of the form. That is, gives the rank of the family of the s symmetric matrices vv^t , where v runs through the minimal vectors.

The algorithm computes the minimal vectors and its runtime is exponential in the dimension of x.

qfrep(q, B, flag)

q being a square and symmetric matrix with integer entries representing a positive definite quadratic form, count the vectors representing successive integers.

- If flag = 0, count all vectors. Outputs the vector whose *i*-th entry, $1 \le i \le B$ is half the number of vectors v such that q(v) = i.
- If flag = 1, count vectors of even norm. Outputs the vector whose *i*-th entry, $1 \le i \le B$ is half the number of vectors such that q(v) = 2i.

```
? q = [2, 1; 1, 3];
? qfrep(q, 5)
%2 = Vecsmall([0, 1, 2, 0, 0]) \\ 1 vector of norm 2, 2 of norm 3, etc.
? qfrep(q, 5, 1)
%3 = Vecsmall([1, 0, 0, 1, 0]) \\ 1 vector of norm 2, 0 of norm 4, etc.
```

This routine uses a naive algorithm based on qfminim, and will fail if any entry becomes larger than 2^{31} (or 2^{63}).

qfsign(x)

Returns [p, m] the signature of the quadratic form represented by the symmetric matrix x. Namely, p (resp. m) is the number of positive (resp. negative) eigenvalues of x. The result is computed using Gaussian reduction.

qfsolve(G)

Given a square symmetric matrix G of dimension $n \ge 1$, solve over \mathbb{Q} the quadratic equation $X^t G X = 0$. The matrix G must have rational coefficients. The solution might be a single nonzero vector (vectorv) or a matrix (whose columns generate a totally isotropic subspace).

If no solution exists, returns an integer, that can be a prime p such that there is no local solution at p, or -1 if there is no real solution, or -2 if n = 2 and $-\det G$ is positive but not a square (which implies there is a real solution, but no local solution at some p dividing det G).

```
? G = [1,0,0;0,1,0;0,0,-34];
? qfsolve(G)
%1 = [-3, -5, 1]~
? qfsolve([1,0; 0,2])
%2 = -1 \\ no real solution
? qfsolve([1,0,0;0,3,0; 0,0,-2])
%3 = 3 \\ no solution in Q_3
? qfsolve([1,0; 0,-2])
%4 = -2 \\ no solution, n = 2
```

quadclassunit(D, flag, tech, precision)

Buchmann-McCurley's sub-exponential algorithm for computing the class group of a quadratic order of discriminant D.

This function should be used instead of qfbclassno or quadregulator when $D < -10^{25}$, $D > 10^{10}$, or when the *structure* is wanted. It is a special case of bnfinit, which is slower, but more robust.

The result is a vector v whose components should be accessed using member functions:

- :math:`v.no`: the class number
- :math: `v.cyc`: a vector giving the structure of the class group as a product of cyclic groups;
- :math: `v.gen`: a vector giving generators of those cyclic groups (as binary quadratic forms).
- :math:`v.reg`: the regulator, computed to an accuracy which is the maximum of an internal accuracy determined by the program and the current default (note that once the regulator is known to a small accuracy it is trivial to compute it to very high accuracy, see the tutorial).

The flag is obsolete and should be left alone. In older versions, it supposedly computed the narrow class group when D > 0, but this did not work at all; use the general function bnfnarrow.

Optional parameter *tech* is a row vector of the form $[c_1, c_2]$, where $c_1 <= c_2$ are nonnegative real numbers which control the execution time and the stack size, see GRHbnf (in the PARI manual). The parameter is used as a threshold to balance the relation finding phase against the final linear algebra. Increasing the default c_1 means that relations are easier to find, but more relations are needed and the linear algebra will be harder. The default value for c_1 is 0 and means that it is taken equal to c_2 . The parameter c_2 is mostly obsolete and should not be changed, but we still document it for completeness: we compute a tentative class group by generators and relations using a factorbase of prime ideals $\langle = c_1(\log ||D||)^2$, then prove that ideals of norm $\langle = c_2(\log ||D||)^2$ do not generate a larger group. By default an optimal c_2 is chosen, so that the result is provably correct under the GRH — a famous result of Bach states that $c_2 = 6$ is fine, but it is possible to improve on this algorithmically. You may provide a smaller c_2 , it will be ignored (we use the provably correct one); you may provide a larger c_2 than the default value, which results in longer computing times for equally correct outputs (under GRH).

quaddisc(x)

Discriminant of the étale algebra $\mathbb{Q}(\sqrt{x})$, where $x \in \mathbb{Q}^*$. This is the same as $\operatorname{coredisc}(d)$ where d is the integer squarefree part of x, so $x = df^2$ with $f \in \mathbb{Q}^*$ and $d \in \mathbb{Z}$. This returns 0 for x = 0, 1 for x square and the discriminant of the quadratic field $\mathbb{Q}(\sqrt{x})$ otherwise.

? quaddisc(7) %1 = 28 ? quaddisc(-7) %2 = -7

quadgen(D, v)

Creates the quadratic number $\omega = (a + \sqrt{D})/2$ where a = 0 if $D = 0 \mod 4$, a = 1 if $D = 1 \mod 4$, so that $(1, \omega)$

is an integral basis for the quadratic order of discriminant D. D must be an integer congruent to 0 or 1 modulo 4, which is not a square. If v is given, the variable name is used to display g else 'w' is used.

```
? w = quadgen(5, 'w); w^2 - w - 1
%1 = 0
? w = quadgen(0, 'w)
*** at top-level: w=quadgen(0)
*** ^____
*** quadgen: domain error in quadpoly: issquare(disc) = 1
```

quadhilbert(D, precision)

Relative equation defining the Hilbert class field of the quadratic field of discriminant D.

If D < 0, uses complex multiplication (Schertz's variant).

If D > 0 Stark units are used and (in rare cases) a vector of extensions may be returned whose compositum is the requested class field. See bnrstark for details.

quadpoly(D, v)

Creates the "canonical" quadratic polynomial (in the variable v) corresponding to the discriminant D, i.e. the minimal polynomial of quadgen(D). D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

```
? quadpoly(5,'y)
%1 = y^2 - y - 1
? quadpoly(0,'y)
*** at top-level: quadpoly(0,'y)
*** ^------
*** quadpoly: domain error in quadpoly: issquare(disc) = 1
```

quadray(D, f, precision)

Relative equation for the ray class field of conductor f for the quadratic field of discriminant D using analytic methods. A bnf for $x^2 - D$ is also accepted in place of D.

For D < 0, uses the σ function and Schertz's method.

For D > 0, uses Stark's conjecture, and a vector of relative equations may be returned. See bnrstark for more details.

quadregulator(x, precision)

Regulator of the quadratic field of positive discriminant x. Returns an error if x is not a discriminant (fundamental or not) or if x is a square. See also quadclassunit if x is large.

quadunit(D, v)

Fundamental unit u of the real quadratic field $\mathbb{Q}(\sqrt{D})$ where D is the positive discriminant of the field. If D is not a fundamental discriminant, this probably gives the fundamental unit of the corresponding order. D must be an integer congruent to 0 or 1 modulo 4, which is not a square; the result is a quadratic number (see quadgen (in the PARI manual)). If v is given, the variable name is used to display u else 'w' is used. The algorithm computes the continued fraction of $(1+\sqrt{D})/2$ or $\sqrt{D}/2$ (see GTM 138, algorithm 5.7.2). Although the continued fraction length is only $O(\sqrt{D})$, the function still runs in time O(D), in part because the output size is not polynomially bounded in terms of log D. See bnfinit and bnfunits for a better alternative for large D, running in time subexponential in log D and returning the fundamental units in compact form (as a short list of S-units of size $O(\log D)^3$ raised to possibly large exponents).

ramanujantau(n)

Compute the value of Ramanujan's tau function at an individual n, assuming the truth of the GRH (to compute quickly class numbers of imaginary quadratic fields using quadclassunit). Algorithm in $O(n^{1/2})$ using

 $O(\log n)$ space. If all values up to N are required, then

$$\sum \tau(n)q^{n} = q \prod_{n>=1} (1-q^{n})^{24}$$

will produce them in time O(N), against $O(N^{3/2})$ for individual calls to ramanujantau; of course the space complexity then becomes O(N).

```
? tauvec(N) = Vec(q*eta(q + 0(q^N))^24);
? N = 10^4; v = tauvec(N);
time = 26 ms.
? ramanujantau(N)
%3 = -482606811957501440000
? w = vector(N, n, ramanujantau(n)); \\ much slower !
time = 13,190 ms.
? v == w
%4 = 1
```

random(N)

Returns a random element in various natural sets depending on the argument N.

- t_INT: returns an integer uniformly distributed between 0 and N 1. Omitting the argument is equivalent to random(2^31).
- t_REAL: returns a real number in [0, 1[with the same accuracy as N (whose mantissa has the same number of significant words).
- t_INTMOD: returns a random intmod for the same modulus.
- t_FFELT: returns a random element in the same finite field.
- t_VEC of length 2, N = [a, b]: returns an integer uniformly distributed between a and b.
- t_VEC generated by ellinit over a finite field k (coefficients are t_INTMOD s modulo a prime or t_FFELT s): returns a "random" k-rational affine point on the curve. More precisely if the curve has a single point (at infinity!) we return it; otherwise we return an affine point by drawing an abscissa uniformly at random until ellordinate succeeds. Note that this is definitely not a uniform distribution over E(k), but it should be good enough for applications.
- t_POL return a random polynomial of degree at most the degree of N. The coefficients are drawn by applying random to the leading coefficient of N.

```
? random(10)
%1 = 9
? random(Mod(0,7))
%2 = Mod(1, 7)
? a = ffgen(ffinit(3,7), 'a); random(a)
%3 = a^6 + 2*a^5 + a^4 + a^3 + a^2 + 2*a
? E = ellinit([3,7]*Mod(1,109)); random(E)
%4 = [Mod(103, 109), Mod(10, 109)]
? E = ellinit([1,7]*a^0); random(E)
%5 = [a^6 + a^5 + 2*a^4 + 2*a^2, 2*a^6 + 2*a^4 + 2*a^3 + a^2 + 2*a]
? random(Mod(1,7)*x^4)
%6 = Mod(5, 7)*x^4 + Mod(6, 7)*x^3 + Mod(2, 7)*x^2 + Mod(2, 7)*x + Mod(5, 7)
```

These variants all depend on a single internal generator, and are independent from your operating system's random number generators. A random seed may be obtained via getrand, and reset using setrand: from a given seed, and given sequence of random s, the exact same values will be generated. The same seed is used at each startup,

reseed the generator yourself if this is a problem. Note that internal functions also call the random number generator; adding such a function call in the middle of your code will change the numbers produced.

Technical note. Up to version 2.4 included, the internal generator produced pseudo-random numbers by means of linear congruences, which were not well distributed in arithmetic progressions. We now use Brent's XORGEN algorithm, based on Feedback Shift Registers, see http://wwwmaths.anu.edu.au/~brent/random.html. The generator has period $2^{4096} - 1$, passes the Crush battery of statistical tests of L'Ecuyer and Simard, but is not suitable for cryptographic purposes: one can reconstruct the state vector from a small sample of consecutive values, thus predicting the entire sequence.

randomprime(N, q)

Returns a strong pseudo prime (see ispseudoprime) in [2, N - 1]. A t_VEC N = [a, b] is also allowed, with $a \le b$ in which case a pseudo prime $a \le p \le b$ is returned; if no prime exists in the interval, the function will run into an infinite loop. If the upper bound is less than 2^{64} the pseudo prime returned is a proven prime.

If the optional parameter q is an integer, return a prime congruent to 1modq; if q is an intmod, return a prime in the given congruence class. If the class contains no prime in the given interval, the function will raise an exception if the class is not invertible, else run into an infinite loop

read(filename)

Reads in the file *filename* (subject to string expansion). If *filename* is omitted, re-reads the last file that was fed into gp. The return value is the result of the last expression evaluated.

If a GP binary file is read using this command (see writebin (in the PARI manual)), the file is loaded and the last object in the file is returned.

In case the file you read in contains an allocatement statement (to be generally avoided), you should leave read instructions by themselves, and not part of larger instruction sequences.

Variants. readvec allows to read a whole file at once; fileopen followed by either fileread (evaluated lines) or filereadstr (lines as nonevaluated strings) allows to read a file one line at a time.

readstr(filename)

Reads in the file *filename* and return a vector of GP strings, each component containing one line from the file. If *filename* is omitted, re-reads the last file that was fed into gp.

readvec(filename)

Reads in the file *filename* (subject to string expansion). If *filename* is omitted, re-reads the last file that was fed into gp. The return value is a vector whose components are the evaluation of all sequences of instructions contained in the file. For instance, if *file* contains

1 2 3

then we will get:

? \r a %1 = 1 %2 = 2 %3 = 3 ? read(a) %4 = 3 ? readvec(a) %5 = [1, 2, 3]

In general a sequence is just a single line, but as usual braces and $\$ may be used to enter multiline sequences.

real(x)

Real part of x. When x is a quadratic number, this is the coefficient of 1 in the "canonical" integral basis $(1, \omega)$.

removeprimes(*x*)

Removes the primes listed in x from the prime number table. In particular removeprimes(addprimes()) empties the extra prime table. x can also be a single integer. List the current extra primes if x is omitted.

rnfalgtobasis(rnf, x)

Expresses x on the relative integral basis. Here, rnf is a relative number field extension L/K as output by rnfinit, and x an element of L in absolute form, i.e. expressed as a polynomial or polmod with polmod coefficients, *not* on the relative integral basis.

rnfbasis(bnf, M)

Let K the field represented by *bnf*, as output by **bnfinit**. M is a projective \mathbb{Z}_K -module of rank n ($M \otimes K$ is an *n*-dimensional K-vector space), given by a pseudo-basis of size n. The routine returns either a true \mathbb{Z}_K -basis of M (of size n) if it exists, or an n + 1-element generating set of M if not.

It is allowed to use a monic irreducible polynomial P in K[X] instead of M, in which case, M is defined as the ring of integers of K[X]/(P), viewed as a \mathbb{Z}_K -module.

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes an order which is maximal at all maximal ideals specified by B, see ??rnfinit: the valuation of D is then correct at all such maximal ideals but may be incorrect at other primes.

rnfbasistoalg(rnf, x)

Computes the representation of x as a polmod with polmods coefficients. Here, rnf is a relative number field extension L/K as output by rnfinit, and x an element of L expressed on the relative integral basis.

rnfcharpoly(nf, T, a, var)

Characteristic polynomial of a over nf, where a belongs to the algebra defined by T over nf, i.e. nf[X]/(T). Returns a polynomial in variable v (x by default).

? nf = nfinit(y^2+1); ? rnfcharpoly(nf, x^2+y*x+1, x+y) %2 = x^2 + Mod(-y, y^2 + 1)*x + 1

rnfconductor(*bnf*, *T*, *flag*)

Given a *bnf* structure attached to a number field K, as produced by bnfinit, and T an irreducible polynomial in K[x] defining an Abelian extension L = K[x]/(T), computes the class field theory conductor of this Abelian extension. If T does not define an Abelian extension over K, the result is undefined; it may be the integer 0 (in which case the extension is definitely not Abelian) or a wrong result.

The result is a 3-component vector [f, bnr, H], where f is the conductor of the extension given as a 2-component row vector $[f_0, f_o o]$, *bnr* is the attached bnr structure and H is a matrix in HNF defining the subgroup of the ray class group on the ray class group generators bnr.gen; in particular, it is a left divisor of the diagonal matrix attached to bnr.cyc and $\|\det H\| = N = \deg T$.

If *flag* is set, return [f, bnrmod, H], where bnrmod is now attached to Cl_f/Cl_f^N , and H is as before since it contains the N-th powers. This is useful when f contains a maximal ideal with huge residue field, since the corresponding tough discrete logarithms are trivialized: in the quotient group, all elements have small order dividing N. This allows to work in Cl_f/H but no longer in Cl_f .

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes the conductor of the extension provided it factors completely over the maximal ideals specified by B, see ??rnfinit: the valuation of f_0 is then correct at all such maximal ideals but may be incorrect at other primes.

rnfdedekind(nf, pol, pr, flag)

Given a number field K coded by nf and a monic polynomial $P \in \mathbb{Z}_K[X]$, irreducible over K and thus defining a relative extension L of K, applies Dedekind's criterion to the order $\mathbb{Z}_K[X]/(P)$, at the prime ideal pr. It is possible to set pr to a vector of prime ideals (test maximality at all primes in the vector), or to omit altogether, in which case maximality at *all* primes is tested; in this situation *flag* is automatically set to 1.

The default historic behavior (*flag* is 0 or omitted and pr is a single prime ideal) is not so useful since **rnfpseudobasis** gives more information and is generally not that much slower. It returns a 3-component vector [max, basis, v]:

- *basis* is a pseudo-basis of an enlarged order O produced by Dedekind's criterion, containing the original order $\mathbb{Z}_K[X]/(P)$ with index a power of *pr*. Possibly equal to the original order.
- *max* is a flag equal to 1 if the enlarged order O could be proven to be *pr*-maximal and to 0 otherwise; it may still be maximal in the latter case if *pr* is ramified in L,
- v is the valuation at pr of the order discriminant.

If *flag* is nonzero, on the other hand, we just return 1 if the order $\mathbb{Z}_K[X]/(P)$ is *pr*-maximal (resp. maximal at all relevant primes, as described above), and 0 if not. This is much faster than the default, since the enlarged order is not computed.

? nf = nfinit(y²-3); P = x³ - 2*y; ? pr3 = idealprimedec(nf,3)[1]; ? rnfdedekind(nf, P, pr3) %3 = [1, [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, 1]], 8] ? rnfdedekind(nf, P, pr3, 1) %4 = 1

In this example, pr3 is the ramified ideal above 3, and the order generated by the cube roots of y is already pr3-maximal. The order-discriminant has valuation 8. On the other hand, the order is not maximal at the prime above 2:

? pr2 = idealprimedec(nf,2)[1]; ? rnfdedekind(nf, P, pr2, 1) %6 = 0 ? rnfdedekind(nf, P, pr2) %7 = [0, [[2, 0, 0; 0, 1, 0; 0, 0, 1], [[1, 0; 0, 1], [1, 0; 0, 1], [1, 1/2; 0, 1/2]]], 2]

The enlarged order is not proven to be pr2-maximal yet. In fact, it is; it is in fact the maximal order:

```
? B = rnfpseudobasis(nf, P)
%8 = [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, [1, 1/2; 0, 1/2]],
[162, 0; 0, 162], -1]
? idealval(nf,B[3], pr2)
%9 = 2
```

It is possible to use this routine with nonmonic $P = \sum_{i < n} p_i X^i \in \mathbb{Z}_K[X]$ if flag = 1; in this case, we test maximality of Dedekind's order generated by

 $1, p_n \alpha, p_n \alpha^2 + p_{n-1} \alpha, \dots, p_n \alpha^{n-1} + p_{n-1} \alpha^{n-2} + \dots + p_1 \alpha.$

The routine will fail if P vanishes on the projective line over the residue field \mathbb{Z}_K/pr (FIXME).

rnfdet(nf, M)

Given a pseudo-matrix M over the maximal order of nf, computes its determinant.

rnfdisc(nf, T)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial $T \in K[x]$ defining a relative extension L = K[x]/(T), compute the relative discriminant of L. This is a vector [D,d], where D is the relative ideal discriminant and d is the relative discriminant considered as an element of K^*/K^{*2} . The main variable of *nf must* be of lower priority than that of T, see priority (in the PARI manual).

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes an order which is maximal at all maximal ideals specified by B, see ??rnfinit: the valuation of D is then correct at all such maximal ideals but may be incorrect at other primes.

rnfeltabstorel(rnf, x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an element of L expressed as a polynomial modulo the absolute equation :emphasis:`rnf.pol`, or in terms of the absolute \mathbb{Z} -basis for \mathbb{Z}_L if rnf contains one (as in rnfinit(nf,pol,1), or after a call to nfinit(rnf)). Computes x as an element of the relative extension L/K as a polmod with polmod coefficients.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? L.polabs
%2 = x^4 + 1
```

rnfeltdown(rnf, x, flag)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L expressed as a polynomial or polmod with polmod coefficients (or as a t_COL on nfinit(rnf).zk), computes x as an element of K as a t_POLMOD if flag = 0 and as a t_COL otherwise. If x is not in K, a domain error occurs.

```
? K = nfinit(y^{2+1}); L = rnfinit(K, x^{2-y});
? L.pol
\%2 = x^{4} + 1
? rnfeltdown(L, Mod(x^2, L.pol))
\%3 = Mod(y, y^2 + 1)
? rnfeltdown(L, Mod(x^2, L.pol), 1)
\%4 = [0, 1] \sim
? rnfeltdown(L, Mod(y, x^2-y))
\%5 = Mod(y, y^2 + 1)
? rnfeltdown(L, Mod(y,K.pol))
\%6 = Mod(y, y^2 + 1)
? rnfeltdown(L, Mod(x, L.pol))
 *** at top-level: rnfeltdown(L,Mod(x,x
 *** rnfeltdown: domain error in rnfeltdown: element not in the base field
? rnfeltdown(L, Mod(y, x^2-y), 1) \\ as a t_COL
\%7 = [0, 1] \sim
? rnfeltdown(L, [0,1,0,0]~) \\ not allowed without absolute nf struct
*** rnfeltdown: incorrect type in rnfeltdown (t_COL).
? nfinit(L); \\ add absolute nf structure to L
? rnfeltdown(L, [0,1,0,0]~) \\ now OK
\%8 = Mod(y, y^2 + 1)
```

If we had started with $L = rnfinit(K, x^2-y, 1)$, then the final would have worked directly.

rnfeltnorm(rnf, x)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L, returns the relative norm $N_{L/K}(x)$ as an element of K.

```
? K = nfinit(y<sup>2</sup>+1); L = rnfinit(K, x<sup>2</sup>-y);
? rnfeltnorm(L, Mod(x, L.pol))
%2 = Mod(x, x<sup>2</sup> + Mod(-y, y<sup>2</sup> + 1))
```

? rnfeltnorm(L, 2)
%3 = 4
? rnfeltnorm(L, Mod(x, x^2-y))

rnfeltreltoabs(rnf, x)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of the absolute extension L/\mathbb{Q} as a polynomial modulo the absolute equation :emphasis: rnf.pol.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? L.pol
%2 = x^4 + 1
? rnfeltreltoabs(L, Mod(x, L.pol))
%3 = Mod(x, x^4 + 1)
? rnfeltreltoabs(L, Mod(y, x^2-y))
%4 = Mod(x^2, x^4 + 1)
? rnfeltreltoabs(L, Mod(y,K.pol))
%5 = Mod(x^2, x^4 + 1)
```

rnfelttrace(rnf, x)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L, returns the relative trace $Tr_{L/K}(x)$ as an element of K.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? rnfelttrace(L, Mod(x, L.pol))
%2 = 0
? rnfelttrace(L, 2)
%3 = 4
? rnfelttrace(L, Mod(x, x^2-y))
```

rnfeltup(rnf, x, flag)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of K, computes x as an element of the absolute extension L/\mathbb{Q} . As a t_POLMOD modulo :emphasis:`rnf.pol` if flag = 0 and as a t_COL on the absolute field integer basis if flag = 1.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? L.pol
%2 = x^4 + 1
? rnfeltup(L, Mod(y, K.pol))
%3 = Mod(x^2, x^4 + 1)
? rnfeltup(L, y)
%4 = Mod(x^2, x^4 + 1)
? rnfeltup(L, [1,2]~) \\ in terms of K.zk
%5 = Mod(2*x^2 + 1, x^4 + 1)
? rnfeltup(L, y, 1) \\ in terms of nfinit(L).zk
%6 = [0, 1, 0, 0]~
? rnfeltup(L, [1,2]~, 1)
%7 = [1, 2, 0, 0]~
```

rnfequation(nf, pol, flag)

Given a number field nf as output by nfinit (or simply a polynomial) and a polynomial *pol* with coefficients in nf defining a relative extension L of nf, computes an absolute equation of L over \mathbb{Q} .

The main variable of nf must be of lower priority than that of pol (see priority (in the PARI manual)). Note that for efficiency, this does not check whether the relative equation is irreducible over nf, but only if it is squarefree. If it is reducible but squarefree, the result will be the absolute equation of the étale algebra defined by pol. If pol is not squarefree, raise an e_DOMAIN exception.

```
? rnfequation(y<sup>2</sup>+1, x<sup>2</sup> - y)
%1 = x<sup>4</sup> + 1
? T = y<sup>3</sup>-2; rnfequation(nfinit(T), (x<sup>3</sup>-2)/(x-Mod(y,T)))
%2 = x<sup>6</sup> + 108 \\ Galois closure of Q(2<sup>(1/3)</sup>)
```

If *flag* is nonzero, outputs a 3-component row vector [z, a, k], where

- z is the absolute equation of L over \mathbb{Q} , as in the default behavior,
- a expresses as a t_POLMOD modulo z a root α of the polynomial defining the base field nf,
- k is a small integer such that θ = β + kα is a root of z, where β is a root of pol. It is guaranteed that k = 0 whenever Q(β) = L.

```
? T = y<sup>3</sup>-2; pol = x<sup>2</sup> +x*y + y<sup>2</sup>;
? [z,a,k] = rnfequation(T, pol, 1);
? z
%3 = x<sup>6</sup> + 108
? subst(T, y, a)
%4 = 0
? alpha= Mod(y, T);
? beta = Mod(x*Mod(1,T), pol);
? subst(z, x, beta + k*alpha)
%7 = 0
```

rnfhnfbasis(*bnf*, *x*)

Given bnf as output by bnfinit, and either a polynomial x with coefficients in bnf defining a relative extension L of bnf, or a pseudo-basis x of such an extension, gives either a true bnf-basis of L in upper triangular Hermite normal form, if it exists, and returns 0 otherwise.

rnfidealabstorel(rnf, x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an ideal of the absolute extension L/\mathbb{Q} . Returns the relative pseudo-matrix in HNF giving the ideal x considered as an ideal of the relative extension L/K, i.e. as a \mathbb{Z}_K -module.

Let Labs be an (absolute) nf structure attached to L, obtained via Labs = nfinit(rnf)). Then rnf "knows" about Labs and x may be given in any format attached to Labs, e.g. a prime ideal or an ideal in HNF wrt. Labs.zk:

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y); Labs = nfinit(rnf);
? m = idealhnf(Labs, 17, x^3+2); \\ some ideal in HNF wrt. Labs.zk
? B = rnfidealabstorel(rnf, m)
%3 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]] \\ pseudo-basis for m as Z_K-module
? A = rnfidealreltoabs(rnf, B)
%4 = [17, x^2 + 4, x + 8, x^3 + 8*x^2] \\ Z-basis for m in Q[x]/(rnf.polabs)
? mathnf(matalgtobasis(Labs, A)) == m
%5 = 1
```

If on the other hand, we do not have a Labs at hand, because it would be too expensive to compute, but we nevertheless have a \mathbb{Z} -basis for x, then we can use the function with this basis as argument. The entries of x may be given either modulo rnf.polabs (absolute form, possibly lifted) or modulo rnf.pol (relative form as t_POLMOD s):

? K = nfinit(y²+1); rnf = rnfinit(K, x²-y); ? rnfidealabstorel(rnf, [17, x² + 4, x + 8, x³ + 8*x²]) %2 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]] ? rnfidealabstorel(rnf, Mod([17, y + 4, x + 8, y*x + 8*y], x²-y)) %3 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]]

rnfidealdown(rnf, x)

Let rnf be a relative number field extension L/K as output by rnfinit, and x an ideal of L, given either in relative form or by a \mathbb{Z} -basis of elements of L (see rnfidealabstorel (in the PARI manual)). This function returns the ideal of K below x, i.e. the intersection of x with K.

rnfidealfactor(rnf, x)

Factor into prime ideal powers the ideal x in the attached absolute number field L = nfinit(rnf). The output format is similar to the factor function, and the prime ideals are represented in the form output by the idealprimedec function for L.

```
? rnf = rnfinit(nfinit(y^2+1), x^2-y+1);
? rnfidealfactor(rnf, y+1) \\ P_2^2
%2 =
[[2, [0,0,1,0]~, 4, 1, [0,0,0,2;0,0,-2,0;-1,-1,0,0;1,-1,0,0]] 2]
? rnfidealfactor(rnf, x) \\ P_2
%3 =
[[2, [0,0,1,0]~, 4, 1, [0,0,0,2;0,0,-2,0;-1,-1,0,0;1,-1,0,0]] 1]
? L = nfinit(rnf);
? id = idealhnf(L, idealhnf(L, 25, (x+1)^2));
? idealfactor(L, id) == rnfidealfactor(rnf, id)
%6 = 1
```

Note that ideals of the base field K must be explicitly lifted to L via **rnfidealup** before they can be factored.

rnfidealhnf(rnf, x)

rnf being a relative number field extension L/K as output by rnfinit and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the HNF pseudo-matrix attached to x, viewed as a \mathbb{Z}_K -module.

rnfidealmul(rnf, x, y)

rnf being a relative number field extension L/K as output by rnfinit and x and y being ideals of the relative extension L/K given by pseudo-matrices, outputs the ideal product, again as a relative ideal.

rnfidealnormabs(rnf, x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal (which can be, as in the absolute case, of many different types, including of course elements). This function computes the norm of the x considered as an ideal of the absolute extension L/\mathbb{Q} . This is identical to

```
idealnorm(rnf, rnfidealnormrel(rnf,x))
```

but faster.

rnfidealnormrel(rnf, x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal (which can be, as in the absolute case, of many different types, including of course elements). This function computes the relative norm of x as an ideal of K in HNF.

rnfidealprimedec(rnf, pr)

Let *rnf* be a relative number field extension L/K as output by rnfinit, and *pr* a maximal ideal of *K* (*prid*), this function completes the *rnf* with a *nf* structure attached to *L* (see rnfinit (in the PARI manual)) and returns the prime ideal decomposition of *pr* in L/K.

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^3+y+1);
? P = idealprimedec(K, 2)[1];
? S = rnfidealprimedec(rnf, P);
? #S
%4 = 1
```

The argument pr is also allowed to be a prime number p, in which case the function returns a pair of vectors [SK, SL], where SK contains the primes of K above p and SL[i] is the vector of primes of L above SK[i].

```
? [SK,SL] = rnfidealprimedec(rnf, 5);
? [#SK, vector(#SL,i,#SL[i])]
%6 = [2, [2, 2]]
```

rnfidealreltoabs(rnf, x, flag)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal, given as a \mathbb{Z}_K -module by a pseudo matrix [A, I]. This function returns the ideal x as an absolute ideal of L/\mathbb{Q} . If flag = 0, the result is given by a vector of t_POLMOD s modulo rnf.pol forming a \mathbb{Z} -basis; if flag = 1, it is given in HNF in terms of the fixed \mathbb{Z} -basis for \mathbb{Z}_L , see rnfinit (in the PARI manual).

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y);
? P = idealprimedec(K,2)[1];
? P = rnfidealup(rnf, P)
%3 = [2, x^2 + 1, 2*x, x^3 + x]
? Prel = rnfidealhnf(rnf, P)
%4 = [[1, 0; 0, 1], [[2, 1; 0, 1], [2, 1; 0, 1]]]
? rnfidealreltoabs(rnf,Prel)
%5 = [2, x^2 + 1, 2*x, x^3 + x]
? rnfidealreltoabs(rnf,Prel,1)
%6 =
[2 1 0 0]
[0 1 0 0]
[0 0 2 1]
[0 0 0 1]
```

The reason why we do not return by default (flag = 0) the customary HNF in terms of a fixed \mathbb{Z} -basis for \mathbb{Z}_L is precisely because a *rnf* does not contain such a basis by default. Completing the structure so that it contains a *nf* structure for *L* is polynomial time but costly when the absolute degree is large, thus it is not done by default. Note that setting flag = 1 will complete the *rnf*.

rnfidealtwoelt(rnf, x)

rnf being a relative number field extension L/K as output by rnfinit and x being an ideal of the relative extension L/K given by a pseudo-matrix, gives a vector of two generators of x over \mathbb{Z}_L expressed as polmods with polmod coefficients.

rnfidealup(rnf, x, flag)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an ideal of K. This function returns the ideal $x\mathbb{Z}_L$ as an absolute ideal of L/\mathbb{Q} , in the form of a \mathbb{Z} -basis. If flag = 0, the result is given by a

vector of polynomials (modulo rnf.pol); if flag = 1, it is given in HNF in terms of the fixed \mathbb{Z} -basis for \mathbb{Z}_L , see rnfinit (in the PARI manual).

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y);
? P = idealprimedec(K,2)[1];
? rnfidealup(rnf, P)
%3 = [2, x^2 + 1, 2*x, x^3 + x]
? rnfidealup(rnf, P,1)
%4 =
[2 1 0 0]
[0 1 0 0]
[0 0 2 1]
[0 0 0 1]
```

The reason why we do not return by default (flag = 0) the customary HNF in terms of a fixed \mathbb{Z} -basis for \mathbb{Z}_L is precisely because a *rnf* does not contain such a basis by default. Completing the structure so that it contains a *nf* structure for *L* is polynomial time but costly when the absolute degree is large, thus it is not done by default. Note that setting flag = 1 will complete the *rnf*.

rnfinit(nf, T, flag)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial T in $\mathbb{Z}_K[x]$ defining a relative extension L = K[x]/(T), this computes data to work in L/K The main variable of T must be of higher priority (see **priority** (in the PARI manual)) than that of nf, and the coefficients of T must be in K.

The result is a row vector, whose components are technical. We let $m = [K : \mathbb{Q}]$ the degree of the base field, n = [L : K] the relative degree, r_1 and r_2 the number of real and complex places of K. Access to this information via *member functions* is preferred since the specific data organization specified below will change in the future.

If flag = 1, add an *nf* structure attached to *L* to *rnf*. This is likely to be very expensive if the absolute degree mn is large, but fixes an integer basis for \mathbb{Z}_L as a \mathbb{Z} -module and allows to input and output elements of *L* in absolute form: as t_COL for elements, as t_MAT in HNF for ideals, as prid for prime ideals. Without such a call, elements of *L* are represented as t_POLMOD, etc. Note that a subsequent nfinit(*rnf*) will also explicitly add such a component, and so will the following functions rnfidealmul, rnfidealtwoelt, rnfidealprimedec, rnfidealup (with flag 1) and rnfidealreltoabs (with flag 1). The absolute *nf* structure attached to *L* can be recovered using nfinit(rnf).

rnf[1]) contains the relative polynomial T.

rnf[2] contains the integer basis [A, d] of K, as (integral) elements of L/\mathbb{Q} . More precisely, A is a vector of polynomial with integer coefficients, d is a denominator, and the integer basis is given by A/d.

rnf[3] (rnf.disc) is a two-component row vector [d(L/K), s] where d(L/K) is the relative ideal discriminant of L/K and s is the discriminant of L/K viewed as an element of $K^*/(K^*)^2$, in other words it is the output of rnfdisc.

rnf[4]) is the ideal index f, i.e. such that $d(T)\mathbb{Z}_K = f^2 d(L/K)$.

rnf[5]) is the list of rational primes dividing the norm of the relative discriminant ideal.

rnf[7] (rnf.zk) is the pseudo-basis (A, I) for the maximal order \mathbb{Z}_L as a \mathbb{Z}_K -module: A is the relative integral pseudo basis expressed as polynomials (in the variable of T) with polmod coefficients in nf, and the second component I is the ideal list of the pseudobasis in HNF.

rnf[8] is the inverse matrix of the integral basis matrix, with coefficients polmods in nf.

rnf[9] is currently unused.

rnf[10] (rnf.nf) is nf.

rnf[11] is an extension of rnfequation(K, T, 1). Namely, a vector [P, a, k, K.pol, T] describing the *absolute* extension L/\mathbb{Q} : P is an absolute equation, more conveniently obtained as rnf.polabs; a expresses the generator $\alpha = ymodK.pol$ of the number field K as an element of L, i.e. a polynomial modulo the absolute equation P;

k is a small integer such that, if β is an abstract root of T and α the generator of K given above, then $P(\beta + k\alpha) = 0$. It is guaranteed that k = 0 if $\mathbb{Q}(\beta) = L$.

Caveat. Be careful if k! = 0 when dealing simultaneously with absolute and relative quantities since $L = \mathbb{Q}(\beta + k\alpha) = K(\alpha)$, and the generator chosen for the absolute extension is not the same as for the relative one. If this happens, one can of course go on working, but we advise to change the relative polynomial so that its root becomes $\beta + k\alpha$. Typical GP instructions would be

```
[P,a,k] = rnfequation(K, T, 1);
if (k, T = subst(T, x, x - k*Mod(y, K.pol)));
L = rnfinit(K, T);
```

rnf[12] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available (which is rarely needed, hence would be too expensive to compute during the initial rnfinit call).

Huge discriminants, helping rnfdisc. When T has a discriminant which is difficult to factor, it is hard to compute \mathbb{Z}_L . As in nfinit, the special input format [T, B] is also accepted, where T is a polynomial as above and B specifies a list of maximal ideals. The following formats are recognized for B:

- an integer: the list of all maximal ideals above a rational prime p < B.
- a vector of rational primes or prime ideals: the list of all maximal ideals dividing an element in the list.

Instead of \mathbb{Z}_L , this produces an order which is maximal at all such maximal ideals primes. The result may actually be a complete and correct *rnf* structure if the relative ideal discriminant factors completely over this list of maximal ideals but this is not guaranteed. In general, the order may not be maximal at primes p not in the list such that p^2 divides the relative ideal discriminant.

rnfisabelian(nf, T)

T being a relative polynomial with coefficients in nf, return 1 if it defines an abelian extension, and 0 otherwise.

```
? K = nfinit(y<sup>2</sup> + 23);
? rnfisabelian(K, x<sup>3</sup> - 3*x - y)
%2 = 1
```

rnfisfree(bnf, x)

Given bnf as output by bnfinit, and either a polynomial x with coefficients in bnf defining a relative extension L of bnf, or a pseudo-basis x of such an extension, returns true (1) if L/bnf is free, false (0) if not.

rnfislocalcyclo(rnf)

Let *rnf* be a relative number field extension L/K as output by **rnfinit** whose degree [L : K] is a power of a prime ℓ . Return 1 if the ℓ -extension is locally cyclotomic (locally contained in the cyclotomic \mathbb{Z}_{ℓ} -extension of K_v at all places $v \| \ell$), and 0 if not.

```
? K = nfinit(y^2 + y + 1);
? L = rnfinit(K, x^3 - y); /* = K(zeta_9), globally cyclotomic */
? rnfislocalcyclo(L)
%3 = 1
\\ we expect 3-adic continuity by Krasner's lemma
```

```
? vector(5, i, rnfislocalcyclo(rnfinit(K, x^3 - y + 3^i)))
%5 = [0, 1, 1, 1, 1]
```

rnfisnorm(T, a, flag)

Similar to bnfisnorm but in the relative case. T is as output by rnfisnorminit applied to the extension L/K. This tries to decide whether the element a in K is the norm of some x in the extension L/K.

The output is a vector [x, q], where a = Norm(x) * q. The algorithm looks for a solution x which is an S-integer, with S a list of places of K containing at least the ramified primes, the generators of the class group of L, as well as those primes dividing a. If L/K is Galois, then this is enough but you may want to add more primes to S to produce different elements, possibly smaller; otherwise, flag is used to add more primes to S: all the places above the primes $p \ll flag$ (resp. p || flag) if flag > 0 (resp. flag < 0).

The answer is guaranteed (i.e. *a* is a norm iff q = 1) if the field is Galois, or, under GRH, if *S* contains all primes less than $12 \log^2 ||\operatorname{disc}(M)||$, where *M* is the normal closure of L/K.

If rnfisnorminit has determined (or was told) that L/K is Galois, and flag! = 0, a Warning is issued (so that you can set flag = 1 to check whether L/K is known to be Galois, according to T). Example:

bnf = bnfinit(y³ + y² - 2*y - 1); p = x² + Mod(y² + 2*y + 1, bnf.pol); T = rnfisnorminit(bnf, p); rnfisnorm(T, 17)

checks whether 17 is a norm in the Galois extension $\mathbb{Q}(\beta)/\mathbb{Q}(\alpha)$, where $\alpha^3 + \alpha^2 - 2\alpha - 1 = 0$ and $\beta^2 + \alpha^2 + 2\alpha + 1 = 0$ (it is).

rnfisnorminit(pol, polrel, flag)

Let K be defined by a root of *pol*, and L/K the extension defined by the polynomial *polrel*. As usual, *pol* can in fact be an *nf*, or *bnf*, etc; if *pol* has degree 1 (the base field is \mathbb{Q}), polrel is also allowed to be an *nf*, etc. Computes technical data needed by **rnfisnorm** to solve norm equations Nx = a, for x in L, and a in K.

If flag = 0, do not care whether L/K is Galois or not.

If flag = 1, L/K is assumed to be Galois (unchecked), which speeds up rnfisnorm.

If flag = 2, let the routine determine whether L/K is Galois.

rnfkummer(bnr, subgp, precision)

This function is deprecated, use bnrclassfield.

rnflllgram(nf, pol, order, precision)

Given a polynomial *pol* with coefficients in *nf* defining a relative extension L and a suborder *order* of L (of maximal rank), as output by rnfpseudobasis(nf, *pol*) or similar, gives [[*neworder*], U], where *neworder* is a reduced order and U is the unimodular transformation matrix.

rnfnormgroup(bnr, pol)

bnr being a big ray class field as output by **bnrinit** and *pol* a relative polynomial defining an Abelian extension, computes the norm group (alias Artin or Takagi group) corresponding to the Abelian extension of bnf = bnr. bnf defined by *pol*, where the module corresponding to *bnr* is assumed to be a multiple of the conductor (i.e. *pol* defines a subextension of bnr). The result is the HNF defining the norm group on the given generators of bnr. gen. Note that neither the fact that *pol* defines an Abelian extension nor the fact that the module is a multiple of the conductor is checked. The result is undefined if the assumption is not correct, but the function will return the empty matrix [;] if it detects a problem; it may also not detect the problem and return a wrong result.

rnfpolred(nf, pol, precision)

This function is obsolete: use rnfpolredbest instead. Relative version of polred. Given a monic polynomial *pol* with coefficients in nf, finds a list of relative polynomials defining some subfields, hopefully simpler and containing the original field. In the present version **2.13.3**, this is slower and less efficient than rnfpolredbest.

Remark. This function is based on an incomplete reduction theory of lattices over number fields, implemented by rnflllgram, which deserves to be improved.

rnfpolredabs(nf, pol, flag)

Relative version of polredabs. Given an irreducible monic polynomial pol with coefficients in the maximal order of nf, finds a canonical relative polynomial defining the same field, hopefully with small coefficients. Note that the equation is only canonical for a fixed nf, using a different defining polynomial in the nf structure will produce a different relative equation.

The binary digits of flag correspond to 1: add information to convert elements to the new representation, 2: absolute polynomial, instead of relative, 16: possibly use a suborder of the maximal order. More precisely:

0: default, return P

1: returns [P, a] where P is the default output and a, a t_POLMOD modulo P, is a root of pol.

2: returns *Pabs*, an absolute, instead of a relative, polynomial. This polynomial is canonical and does not depend on the *nf* structure. Same as but faster than

polredabs(rnfequation(nf, pol))

3: returns [Pabs, a, b], where *Pabs* is an absolute polynomial as above, a, b are t_POLMOD modulo *Pabs*, roots of nf.pol and *pol* respectively.

16: (OBSOLETE) possibly use a suborder of the maximal order. This makes rnfpolredabs behave as rnfpolredbest. Just use the latter.

Warning. The complexity of rnfpolredabs is exponential in the absolute degree. The function rnfpolredbest runs in polynomial time, and tends to return polynomials with smaller discriminants. It also supports polynomials with arbitrary coefficients in *nf*, neither integral nor necessarily monic.

rnfpolredbest(nf, pol, flag)

Relative version of polredbest. Given a polynomial *pol* with coefficients in nf, finds a simpler relative polynomial P defining the same field. As opposed to rnfpolredabs this function does not return a *smallest* (canonical) polynomial with respect to some measure, but it does run in polynomial time.

The binary digits of flag correspond to 1: add information to convert elements to the new representation, 2: absolute polynomial, instead of relative. More precisely:

0: default, return P

1: returns [P, a] where P is the default output and a, a t_POLMOD modulo P, is a root of pol.

2: returns Pabs, an absolute, instead of a relative, polynomial. Same as but faster than

rnfequation(nf, rnfpolredbest(nf,pol))

3: returns [Pabs, a, b], where *Pabs* is an absolute polynomial as above, a, b are t_POLMOD modulo *Pabs*, roots of nf.pol and *pol* respectively.

```
? K = nfinit(y<sup>3</sup>-2); pol = x<sup>2</sup> +x*y + y<sup>2</sup>;
? [P, a] = rnfpolredbest(K,pol,1);
? P
%3 = x<sup>2</sup> - x + Mod(y - 1, y<sup>3</sup> - 2)
```

```
? a
%4 = Mod(Mod(2*y^{2}+3*y+4,y^{3}-2)*x + Mod(-y^{2}-2*y-2,y^{3}-2),
x^2 - x + Mod(y-1, y^3-2))
? subst(K.pol,y,a)
%5 = 0
? [Pabs, a, b] = rnfpolredbest(K,pol,3);
? Pabs
\%7 = x^{6} - 3^{*}x^{5} + 5^{*}x^{3} - 3^{*}x + 1
? a
\%8 = Mod(-x^2+x+1, x^6-3*x^5+5*x^3-3*x+1)
?b
\%9 = Mod(2*x^{5}-5*x^{4}-3*x^{3}+10*x^{2}+5*x^{5}, x^{6}-3*x^{5}+5*x^{3}-3*x^{1})
? subst(K.pol,y,a)
\%10 = 0
? substvec(pol,[x,y],[a,b])
\%11 = 0
```

rnfpseudobasis(nf, T)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial Tin $\mathbb{Z}_K[x]$ defining a relative extension L = K[x]/(T), computes the relative discriminant of L and a pseudo-basis (A, J) for the maximal order \mathbb{Z}_L viewed as a \mathbb{Z}_K -module. This is output as a vector [A, J, D, d], where D is the relative ideal discriminant and d is the relative discriminant considered as an element of K^*/K^{*2} .

```
? K = nfinit(y^2+1);
? [A,J,D,d] = rnfpseudobasis(K, x^2+y);
? A
%3 =
[1 0]
[0 1]
? J
%4 = [1, 1]
? D
%5 = [0, -4]~
? d
%6 = [0, -1]~
```

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and produce an order which is maximal at all prime ideals specified by B, see ??rnfinit.

```
? p = 585403248812100232206609398101;
? q = 711171340236468512951957953369;
? T = x^2 + 3*(p*q)^2;
? [A,J,D,d] = V = rnfpseudobasis(K, T); D
time = 22,178 ms.
%10 = 3
? [A,J,D,d] = W = rnfpseudobasis(K, [T,100]); D
time = 5 ms.
%11 = 3
? V == W
%12 = 1
```

```
? [A,J,D,d] = W = rnfpseudobasis(K, [T, [3]]); D
%13 = 3
? V == W
%14 = 1
```

In this example, the results are identical since $D \cap \mathbb{Z}$ factors over primes less than 100 (and in fact, over 3). Had it not been the case, the order would have been guaranteed maximal at primes p || p for $p \leq 100$ only (resp. p || 3). And might have been nonmaximal at any other prime ideal p such that p^2 divided D.

rnfsteinitz(nf, x)

Given a number field nf as output by nfinit and either a polynomial x with coefficients in nf defining a relative extension L of nf, or a pseudo-basis x of such an extension as output for example by rnfpseudobasis, computes another pseudo-basis (A, I) (not in HNF in general) such that all the ideals of I except perhaps the last one are equal to the ring of integers of nf, and outputs the four-component row vector [A, I, D, d] as in rnfpseudobasis. The name of this function comes from the fact that the ideal class of the last ideal of I, which is well defined, is the Steinitz class of the \mathbb{Z}_K -module \mathbb{Z}_L (its image in $SK_0(\mathbb{Z}_K)$).

rootsof1(N, precision)

Return the column vector v of all complex N-th roots of 1, where N is a positive integer. In other words, $v[k] = \exp(2I(k-1)\pi/N)$ for k = 1, ..., N. Rational components (e.g., the roots 1 and I) are given exactly, not as floating point numbers:

? rootsof1(4)
%1 = [1, I, -1, -I]~
? rootsof1(3)
%2 = [1, -1/2 + 0.866025...*I, -1/2 - 0.866025...*I]~

round(x, e)

If x is in \mathbb{R} , rounds x to the nearest integer (rounding to +oo in case of ties), then and sets e to the number of error bits, that is the binary exponent of the difference between the original and the rounded value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. e > 0), the result is undefined and an error occurs if e was not given.

Important remark. Contrary to the other truncation functions, this function operates on every coefficient at every level of a PARI object. For example

$$truncate((2.4 * X^2 - 1.7)/(X)) = 2.4 * X,$$

whereas

$$round((2.4 * X^2 - 1.7)/(X)) = (2 * X^2 - 2)/(X).$$

An important use of **round** is to get exact results after an approximate computation, when theory tells you that the coefficients must be integers.

select(f, A, flag)

We first describe the default behavior, when flag is 0 or omitted. Given a vector or list A and a t_CLOSURE f, select returns the elements x of A such that f(x) is nonzero. In other words, f is seen as a selection function returning a boolean value.

```
? select(x->isprime(x), vector(50,i,i^2+1))
%1 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]
? select(x->(x<100), %)
%2 = [2, 5, 17, 37]</pre>
```

returns the primes of the form $i^2 + 1$ for some $i \le 50$, then the elements less than 100 in the preceding result. The select function also applies to a matrix A, seen as a vector of columns, i.e. it selects columns instead of entries, and returns the matrix whose columns are the selected ones.

Remark. For v a t_VEC, t_COL, t_VECSMALL, t_LIST or t_MAT, the alternative set-notations

[g(x) | x <- v, f(x)] [x | x <- v, f(x)] [g(x) | x <- v]

are available as shortcuts for

```
apply(g, select(f, Vec(v)))
select(f, Vec(v))
apply(g, Vec(v))
```

respectively:

? [x | x <- vector(50,i,i^2+1), isprime(x)]
%1 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]</pre>

If flag = 1, this function returns instead the *indices* of the selected elements, and not the elements themselves (indirect selection):

```
? V = vector(50,i,i^2+1);
? select(x->isprime(x), V, 1)
%2 = Vecsmall([1, 2, 4, 6, 10, 14, 16, 20, 24, 26, 36, 40])
? vecextract(V, %)
%3 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]
```

The following function lists the elements in $(\mathbb{Z}/N\mathbb{Z})^*$:

? invertibles(N) = select($x \rightarrow gcd(x, N) == 1, [1..N]$)

Finally

? select($x \rightarrow x$, M)

selects the nonzero entries in M. If the latter is a t_MAT , we extract the matrix of nonzero columns. Note that *removing* entries instead of selecting them just involves replacing the selection function f with its negation:

? select(x->!isprime(x), vector(50,i,i^2+1))

self()

Return the calling function or closure as a t_CLOSURE object. This is useful for defining anonymous recursive functions.

```
? (n -> if(n==0,1,n*self()(n-1)))(5)
%1 = 120 \\ 5!
? (n -> if(n<=1, n, self()(n-1)+self()(n-2)))(20)
%2 = 6765 \\ Fibonacci(20)
```

seralgdep(s, p, r)

finds a linear relation between powers $(1, s, ..., s^p)$ of the series s, with polynomial coefficients of degree $\langle = r$. In case no relation is found, return 0.

```
? s = 1 + 10*y - 46*y^2 + 460*y^3 - 5658*y^4 + 77740*y^5 + 0(y^6);

? seralgdep(s, 2, 2)

%2 = -x^2 + (8*y^2 + 20*y + 1)

? subst(%, x, s)

%3 = 0(y^6)

? seralgdep(s, 1, 3)

%4 = (-77*y^2 - 20*y - 1)*x + (310*y^3 + 231*y^2 + 30*y + 1)

? seralgdep(s, 1, 2)

%5 = 0
```

The series main variable must not be x, so as to be able to express the result as a polynomial in x.

serchop(s, n)

Remove all terms of degree strictly less than n in series s. When the series contains no terms of degree < n, return $O(x^n)$.

? s = 1/x + x + 2*x² + 0(x³); ? serchop(s) %2 = x + 2*x³ + 0(x³) ? serchop(s, 2) %3 = 2*x² + 0(x³) ? serchop(s, 100) %4 = 0(x¹00)

serconvol(x, y)

Convolution (or Hadamard product) of the two power series x and y; in other words if $x = \sum a_k * X^k$ and $y = \sum b_k * X^k$ then $serconvol(x, y) = \sum a_k * b_k * X^k$.

serlaplace(x)

x must be a power series with nonnegative exponents or a polynomial. If $x = \sum (a_k/k!) * X^k$ then the result is $\sum a_k * X^k$.

serprec(x, v)

Returns the absolute precision of x with respect to power series in the variable v; this is the minimum precision of the components of x. The result is +oo if x is an exact object (as a series in v):

```
? serprec(x + 0(y^2), y)
%1 = 2
? serprec(x + 2, x)
%2 = +oo
? serprec(2 + x + 0(x^2), y)
%3 = +oo
```

serreverse(s)

Reverse power series of s, i.e. the series t such that t(s) = x; s must be a power series whose valuation is exactly equal to one.

? \ps 8 ? t = serreverse(tan(x)) %2 = x - $1/3*x^3 + 1/5*x^5 - 1/7*x^7 + 0(x^8)$? tan(t) %3 = x + $0(x^8)$

setbinop(f, X, Y)

The set whose elements are the f(x,y), where x,y run through X,Y. respectively. If Y is omitted, assume that X = Y and that f is symmetric: f(x, y) = f(y, x) for all x, y in X.

? X = [1,2,3]; Y = [2,3,4]; ? setbinop((x,y)->x+y, X,Y) \\ set X + Y %2 = [3, 4, 5, 6, 7] ? setbinop((x,y)->x-y, X,Y) \\ set X - Y %3 = [-3, -2, -1, 0, 1] ? setbinop((x,y)->x+y, X) \\ set 2X = X + X %2 = [2, 3, 4, 5, 6]

setintersect(x, y)

Intersection of the two sets x and y (see setisset). If x or y is not a set, the result is undefined.

setisset(x)

Returns true (1) if x is a set, false (0) if not. In PARI, a set is a row vector whose entries are strictly increasing with respect to a (somewhat arbitrary) universal comparison function. To convert any object into a set (this is most useful for vectors, of course), use the function Set.

? a = [3, 1, 1, 2]; ? setisset(a) %2 = 0 ? Set(a) %3 = [1, 2, 3]

setminus(x, y)

Difference of the two sets x and y (see setisset), i.e. set of elements of x which do not belong to y. If x or y is not a set, the result is undefined.

setrand(n)

Reseeds the random number generator using the seed n. No value is returned. The seed is a small positive integer $0 < n < 2^{64}$ used to generate deterministically a suitable state array. All gp session start by an implicit setrand(1), so resetting the seed to this value allows to replay all computations since the session start. Alternatively, running a randomized computation starting by setrand (n) twice with the same n will generate the exact same output.

In the other direction, including a call to setrand(getwalltime()) from your gprc will cause GP to produce different streams of random numbers in each session. (Unix users may want to use /dev/urandom instead of getwalltime.)

For debugging purposes, one can also record a particular random state using getrand (the value is encoded as a huge integer) and feed it to setrand:

```
? state = getrand(); \\ record seed
...
? setrand(state); \\ we can now replay the exact same computations
```

setsearch(S, x, flag)

Determines whether x belongs to the set S (see setisset).

We first describe the default behavior, when flag is zero or omitted. If x belongs to the set S, returns the index j such that S[j] = x, otherwise returns 0.

? T = [7,2,3,5]; S = Set(T); ? setsearch(S, 2) %2 = 1 ? setsearch(S, 4) \\ not found %3 = 0 ? setsearch(T, 7) \\ search in a randomly sorted vector %4 = 0 \\ WRONG !

If S is not a set, we also allow sorted lists with respect to the cmp sorting function, without repeated entries, as per listsort(L, 1); otherwise the result is undefined.

```
? L = List([1,4,2,3,2]); setsearch(L, 4)
%1 = 0 \\ WRONG !
? listsort(L, 1); L \\ sort L first
%2 = List([1, 2, 3, 4])
? setsearch(L, 4)
%3 = 4 \\ now correct
```

If flag is nonzero, this function returns the index j where x should be inserted, and 0 if it already belongs to S. This is meant to be used for dynamically growing (sorted) lists, in conjunction with listinsert.

? L = List([1,5,2,3,2]); listsort(L,1); L %1 = List([1,2,3,5]) ? j = setsearch(L, 4, 1) \\ 4 should have been inserted at index j %2 = 4 ? listinsert(L, 4, j); L %3 = List([1, 2, 3, 4, 5])

setunion(x, y)

Union of the two sets x and y (see setisset). If x or y is not a set, the result is undefined.

shift(x, n)

Shifts x componentwise left by n bits if $n \ge 0$ and right by ||n|| bits if n < 0. May be abbreviated as x :literal: <<` n or x :literal:` >>` (-n). A left shift by n corresponds to multiplication by 2^n . A right shift of an integer x by ||n|| corresponds to a Euclidean division of x by $2^{||n||}$ with a remainder of the same sign as x, hence is not the same (in general) as $x 2^n$.

shiftmul(x, n)

Multiplies x by 2^n . The difference with shift is that when n < 0, ordinary division takes place, hence for example if x is an integer the result may be a fraction, while for shifts Euclidean division takes place when n < 0 hence if x is an integer the result is still an integer.

sigma(x, k)

Sum of the k - th powers of the positive divisors of ||x||. x and k must be of type integer.

sign(x)

sign (0, 1 or -1) of x, which must be of type integer, real or fraction; t_QUAD with positive discriminants and t_INFINITY are also supported.

simplify(x)

This function simplifies x as much as it can. Specifically, a complex or quadratic number whose imaginary part is the integer 0 (i.e. not Mod(0, 2) or 0.E-28) is converted to its real part, and a polynomial of degree 0 is converted to its constant term. Simplifications occur recursively.

This function is especially useful before using arithmetic functions, which expect integer arguments:

? x = 2 + y - y %1 = 2 ? isprime(x) *** at top-level: isprime(x) *** ^------*** isprime: not an integer argument in an arithmetic function ? type(x) %2 = "t_POL" ? type(simplify(x)) %3 = "t_INT"

Note that GP results are simplified as above before they are stored in the history. (Unless you disable automatic simplification with \backslash y, that is.) In particular

? type(%1) %4 = "t_INT"

sin(x, precision)

Sine of x. Note that, for real x, cosine and sine can be obtained simultaneously as

 $cs(x) = my(z = exp(I^*x)); [real(z), imag(z)];$

and for general complex x as

cs2(x) = my(z = exp(I*x), u = 1/z); [(z+u)/2, (z-u)/2];

Note that the latter function suffers from catastrophic cancellation when z^2 1.

sinc(x, precision)

Cardinal sine of x, i.e. $\sin(x)/x$ if x! = 0, 1 otherwise. Note that this function also allows to compute

$$(1 - \cos(x))/x^2 = sinc(x/2)^2/2$$

accurately near x = 0.

sinh(x, precision)

Hyperbolic sine of x.

sizebyte(x)

Outputs the total number of bytes occupied by the tree representing the PARI object x.

sizedigit(x)

This function is DEPRECATED, essentially meaningless, and provided for backwards compatibility only. Don't use it!

outputs a quick upper bound for the number of decimal digits of (the components of) x, off by at most 1. More precisely, for a positive integer x, it computes (approximately) the ceiling of

$$floor(1 + \log_2 x) \log_{10} 2$$

To count the number of decimal digits of a positive integer x, use #digits(x). To estimate (recursively) the size of x, use normlp(x).

sqr(x)

Square of x. This operation is not completely straightforward, i.e. identical to x * x, since it can usually be computed more efficiently (roughly one-half of the elementary multiplications can be saved). Also, squaring a 2-adic number increases its precision. For example,

? $(1 + 0(2^4))^2$ %1 = 1 + 0(2^5) ? $(1 + 0(2^4)) * (1 + 0(2^4))$ %2 = 1 + 0(2^4)

Note that this function is also called whenever one multiplies two objects which are known to be *identical*, e.g. they are the value of the same variable, or we are computing a power.

? x = (1 + 0(2⁴)); x * x %3 = 1 + 0(2⁵) ? (1 + 0(2⁴))⁴ %4 = 1 + 0(2⁶)

(note the difference between %2 and %3 above).

sqrt(x, precision)

Principal branch of the square root of x, defined as $\sqrt{x} = \exp(\log x/2)$. In particular, we have $Arg(sqrt(x)) \in [-\pi/2, \pi/2]$, and if $x \in \mathbb{R}$ and x < 0, then the result is complex with positive imaginary part.

Intmod a prime p, t_PADIC and t_FFELT are allowed as arguments. In the first 2 cases (t_INTMOD, t_PADIC), the square root (if it exists) which is returned is the one whose first p-adic digit is in the interval [0, p/2]. For other arguments, the result is undefined.

sqrtint(x, r)

Returns the integer square root of x, i.e. the largest integer y such that $y^2 \le x$, where x a nonnegative integer. If r is present, set it to the remainder $r = x - y^2$, which satisfies $0 \le r \le 2y$

```
? x = 120938191237; sqrtint(x)
%1 = 347761
? sqrt(x)
%2 = 347761.68741970412747602130964414095216
? y = sqrtint(x, &r)
%3 = 347761
? x - y^2
%4 = 478116
```

sqrtn(x, n, z, precision)

Principal branch of the *n*, i.e. such that $Arg(sqrtn(x)) \in]-\pi/n, \pi/n]$. Intmod a prime and *p*-adics are allowed as arguments.

If z is present, it is set to a suitable root of unity allowing to recover all the other roots. If it was not possible, z is set to zero. In the case this argument is present and no n is returned instead of raising an error.

```
? sqrtn(Mod(2,7), 2)
%1 = Mod(3, 7)
? sqrtn(Mod(2,7), 2, &z); z
%2 = Mod(6, 7)
? sqrtn(Mod(2,7), 3)
*** at top-level: sqrtn(Mod(2,7),3)
*** ^------
*** sqrtn: nth-root does not exist in gsqrtn.
? sqrtn(Mod(2,7), 3, &z)
%2 = 0
? z
%3 = 0
```

The following script computes all roots in all possible cases:

```
sqrtnall(x,n)=
{ my(V,r,z,r2);
    r = sqrtn(x,n, &z);
    if (!z, error("Impossible case in sqrtn"));
    if (type(x) == "t_INTMOD" || type(x)=="t_PADIC",
    r2 = r*z; n = 1;
    while (r2!=r, r2*=z;n++));
    V = vector(n); V[1] = r;
    for(i=2, n, V[i] = V[i-1]*z);
    V
}
addhelp(sqrtnall,"sqrtnall(x,n):compute the vector of nth-roots of x");
```

sqrtnint(x, n)

Returns the integer n-th root of x, i.e. the largest integer y such that $y^n \le x$, where x is a nonnegative integer.

? N = 120938191237; sqrtnint(N, 5) %1 = 164 ? N^(1/5) %2 = 164.63140849829660842958614676939677391

The special case n = 2 is sqrtint

stirling(n, k, flag)

Stirling number of the first kind s(n, k) (flag = 1, default) or of the second kind S(n, k) (flag = 2), where n, k are nonnegative integers. The former is $(-1)^{n-k}$ times the number of permutations of n symbols with exactly k cycles; the latter is the number of ways of partitioning a set of n elements into k nonempty subsets. Note that if all s(n, k) are needed, it is much faster to compute

$$\sum_{k} s(n,k)x^{k} = x(x-1)...(x-n+1).$$

Similarly, if a large number of S(n, k) are needed for the same k, one should use

$$\sum_{n} S(n,k)x^{n} = (x^{k})/((1-x)...(1-kx)).$$

(Should be implemented using a divide and conquer product.) Here are simple variants for n fixed:

```
/* list of s(n,k), k = 1..n */
vecstirling(n) = Vec( factorback(vector(n-1,i,1-i*'x)) )
/* list of S(n,k), k = 1..n */
vecstirling2(n) =
{ my(Q = x^(n-1), t);
vector(n, i, t = divrem(Q, x-i); Q=t[1]; simplify(t[2]));
}
/* Bell numbers, B_n = B[n+1] = sum(k = 0, n, S(n,k)), n = 0..N */
vecbell(N)=
{ my (B = vector(N+1));
B[1] = B[2] = 1;
for (n = 2, N,
```

```
my (C = binomial(n-1));
B[n+1] = sum(k = 1, n, C[k]*B[k]);
); B;
}
```

strchr(x)

Converts integer or vector of integers x to a string, translating each integer (in the range [1, 255]) into a character using ASCII encoding.

? strchr(97)
%1 = "a"
? Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100])
? strchr(%)
%3 = "hello world"

strjoin(v, p)

Joins the strings in vector v, separating them with delimiter p. The reverse operation is strsplit.

? v = ["abc", "def", "ghi"]
? strjoin(v, "/")
%2 = "abc/def/ghi"
? strjoin(v)
%3 = "abcdefghi"

strsplit(s, p)

Splits the string s into a vector of strings, with p acting as a delimiter. If p is empty or omitted, split the string into characters.

? strsplit("abc::def::ghi", "::")
%1 = ["abc", "def", "ghi"]
? strsplit("abc", "")
%2 = ["a", "b", "c"]
? strsplit("aba", "a")

If s starts (resp. ends) with the pattern p, then the first (resp. last) entry in the vector is the empty string:

```
? strsplit("aba", "a")
%3 = ["", "b", ""]
```

strtime(t)

Return a string describing the time t in milliseconds in the format used by the GP timer.

```
? print(strtime(12345678))
3h, 25min, 45,678 ms
? {
    my(t=getabstime());
    F=factor(2^256+1);t=getabstime()-t;
    print("factor(2^256+1) took ",strtime(t));
    }
factor(2^256+1) took 1,320 ms
```

subgrouplist(cyc, bound, flag)

cyc being a vector of positive integers giving the cyclic components for a finite Abelian group G (or any object which has a .cyc method), outputs the list of subgroups of G. Subgroups are given as HNF left divisors of the SNF matrix corresponding to G.

If flag = 0 (default) and *cyc* is a *bnr* structure output by bnrinit, gives only the subgroups whose modulus is the conductor. Otherwise, all subgroups are given.

If *bound* is present, and is a positive integer, restrict the output to subgroups of index less than *bound*. If *bound* is a vector containing a single positive integer B, then only subgroups of index exactly equal to B are computed. For instance

```
? subgrouplist([6,2])
%1 = [[6, 0; 0, 2], [2, 0; 0, 2], [6, 3; 0, 1], [2, 1; 0, 1], [3, 0; 0, 2],
[1, 0; 0, 2], [6, 0; 0, 1], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],3) \\ index less than 3
%2 = [[2, 1; 0, 1], [1, 0; 0, 2], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],[3]) \\ index 3
%3 = [[3, 0; 0, 1]]
? bnr = bnrinit(bnfinit(x), [120,[1]], 1);
? L = subgrouplist(bnr, [8]);
```

In the last example, L corresponds to the 24 subfields of $\mathbb{Q}(\zeta_{120})$, of degree 8 and conductor 12000 (by setting *flag*, we see there are a total of 43 subgroups of degree 8).

? vector(#L, i, galoissubcyclo(bnr, L[i]))

will produce their equations. (For a general base field, you would have to rely on bnrstark, or bnrclassfield.)

Warning. This function requires factoring the exponent of G. If you are only interested in subgroups of index n (or dividing n), you may considerably speed up the function by computing the subgroups of G/G^n , whose cyclic components are apply(x - > gcd(n,x), C) (where C gives the cyclic components of G). If you want the *bnr* variant, now is a good time to use bnrinit(,,, n) as well, to directly compute the ray class group modulo n-th powers.

subst(x, y, z)

Replace the simple variable y by the argument z in the "polynomial" expression x. If z is a vector, return the vector of the evaluated expressions subst(x, y, z[i]).

Every type is allowed for x, but if it is not a genuine polynomial (or power series, or rational function), the substitution will be done as if the scalar components were polynomials of degree zero. In particular, beware that:

```
? subst(1, x, [1,2; 3,4])
%1 =
[1 0]
[0 1]
? subst(1, x, Mat([0,1]))
 *** at top-level: subst(1,x,Mat([0,1])
 *** ^-----
*** subst: forbidden substitution by a non square matrix.
```

If x is a power series, z must be either a polynomial, a power series, or a rational function. If x is a vector, matrix or list, the substitution is applied to each individual entry.

Use the function substvec to replace several variables at once, or the function substpol to replace a polynomial expression.

substpol(x, y, z)

Replace the "variable" y by the argument z in the "polynomial" expression x. Every type is allowed for x, but the same behavior as subst above apply.

The difference with subst is that y is allowed to be any polynomial here. The substitution is done moding out all components of x (recursively) by y - t, where t is a new free variable of lowest priority. Then substituting t by z in the resulting expression. For instance

```
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, x<sup>2</sup>, y)
%1 = y<sup>2</sup> + y + 1
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, x<sup>3</sup>, y)
%2 = x<sup>2</sup> + y<sup>*</sup>x + 1
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, (x+1)<sup>2</sup>, y)
%3 = (-4<sup>*</sup>y - 6)<sup>*</sup>x + (y<sup>2</sup> + 3<sup>*</sup>y - 3)
```

substvec(x, v, w)

v being a vector of monomials of degree 1 (variables), w a vector of expressions of the same length, replace in the expression x all occurrences of v_i by w_i . The substitutions are done simultaneously; more precisely, the v_i are first replaced by new variables in x, then these are replaced by the w_i :

? substvec([x,y], [x,y], [y,x])
%1 = [y, x]
? substvec([x,y], [x,y], [y,x+y])
%2 = [y, x + y] \\ not [y, 2*y]

sumdedekind(h, k)

Returns the Dedekind sum attached to the integers h and k, corresponding to a fast implementation of

s(h,k) = sum(n = 1, k-1, (n/k)*(frac(h*n/k) - 1/2))

sumdigits(n, B)

Sum of digits in the integer ||n||, when written in base B > 1.

```
? sumdigits(123456789)
%1 = 45
? sumdigits(123456789, 2)
%1 = 16
```

Note that the sum of bits in n is also returned by hammingweight. This function is much faster than vecsum(digits(n,B)) when B is 10 or a power of 2, and only slightly faster in other cases.

sumeulerrat(F, s, a, precision)

 $\sum_{n>=a} F(p^s)$, where the sum is taken over prime numbers and F is a rational function.

```
? sumeulerrat(1/p^2)
%1 = 0.45224742004106549850654336483224793417
? sumeulerrat(1/p, 2)
%2 = 0.45224742004106549850654336483224793417
```

sumformal(f, v)

formal sum of the polynomial expression f with respect to the main variable if v is omitted, with respect to the variable v otherwise; it is assumed that the base ring has characteristic zero. In other words, considering f as

a polynomial function in the variable v, returns F, a polynomial in v vanishing at 0, such that $F(b) - F(a) = sum_{v=a+1}^{b} f(v)$:

```
? sumformal(n) \setminus 1 + ... + n

\%1 = 1/2*n^2 + 1/2*n

? f(n) = n^3+n^2+1;

? F = sumformal(f(n)) \setminus f(1) + ... + f(n)

\%3 = 1/4*n^4 + 5/6*n^3 + 3/4*n^2 + 7/6*n

? sum(n = 1, 2000, f(n)) == subst(F, n, 2000)

\%4 = 1

? sum(n = 1001, 2000, f(n)) == subst(F, n, 2000) - subst(F, n, 1000)

\%5 = 1

? sumformal(x^2 + x*y + y^2, y)

\%6 = y*x^2 + (1/2*y^2 + 1/2*y)*x + (1/3*y^3 + 1/2*y^2 + 1/6*y)

? x^2 * y + x * sumformal(y) + sumformal(y^2) == %

\%7 = 1
```

sumnumapinit(asymp, precision)

Initialize tables for Abel-Plana summation of a series $\sum f(n)$, where f is holomorphic in a right half-plane. If given, asymp is of the form $[+oo, \alpha]$, as in intrum and indicates the decrease rate at infinity of functions to be summed. A positive $\alpha > 0$ encodes an exponential decrease of type $\exp(-\alpha n)$ and a negative $-2 < \alpha < -1$ encodes a slow polynomial decrease of type n^{α} .

```
? \p200
? sumnumap(n=1, n^{-2});
time = 163 \text{ ms}.
? tab = sumnumapinit();
time = 160 \text{ ms.}
? sumnumap(n=1, n^-2, tab); \\ faster
time = 7 \text{ ms.}
? tab = sumnumapinit([+oo, log(2)]); \\ decrease like 2^-n
time = 164 \text{ ms}.
? sumnumap(n=1, 2^-n, tab) - 1
time = 36 \text{ ms}.
%5 = 3.0127431466707723218 E-282
? tab = sumnumapinit([+oo, -4/3]); \\ decrease like n^(-4/3)
time = 166 \text{ ms.}
? sumnumap(n=1, n^(-4/3), tab);
time = 181 \text{ ms.}
```

sumnuminit(asymp, precision)

Initialize tables for Euler-MacLaurin delta summation of a series with positive terms. If given, asymp is of the form $[+oo, \alpha]$, as in intrum and indicates the decrease rate at infinity of functions to be summed. A positive $\alpha > 0$ encodes an exponential decrease of type $\exp(-\alpha n)$ and a negative $-2 < \alpha < -1$ encodes a slow polynomial decrease of type n^{α} .

```
? \p200
? sumnum(n=1, n^-2);
time = 200 ms.
? tab = sumnuminit();
time = 188 ms.
```

```
? sumnum(n=1, n^-2, tab); \\ faster
time = 8 ms.
? tab = sumnuminit([+oo, log(2)]); \\ decrease like 2^-n
time = 200 ms.
? sumnum(n=1, 2^-n, tab)
time = 44 ms.
? tab = sumnuminit([+oo, -4/3]); \\ decrease like n^(-4/3)
time = 200 ms.
? sumnum(n=1, n^(-4/3), tab);
time = 221 ms.
```

sumnumlagrangeinit(asymp, c1, precision)

Initialize tables for Lagrange summation of a series. By default, assume that the remainder $R(n) = \sum_{m>=n} f(m)$ has an asymptotic expansion

$$R(n) = \sum_{m>=n} f(n) \sum_{i>=1} a_i / n^i$$

at infinity. The argument asymp allows to specify different expansions:

• a real number β means

$$R(n) = n^{-\beta} \sum_{i>=1} a_i / n^i$$

a t_CLOSURE g means

$$R(n) = g(n) \sum_{i>=1} a_i/n^i$$
 (The
preceding case corresponds to : math : 'g(n) = n^{-\beta} .)

• a pair $[\alpha, \beta]$ where β is as above and $\alpha \in 2, 1, 1/2, 1/3, 1/4$. We let $R_2(n) = R(n) - f(n)/2$ and $R_\alpha(n) = R(n)$ for $\alpha! = 2$. Then

$$R_{\alpha}(n) = g(n) \sum_{i>=1} a_i / n^{i\alpha}$$

 $Note that the initialization times increase considerable for the : math : `\alpha` is this list(: math : `1/4` being the slowest).$

The constant c1 is technical and computed by the program, but can be set by the user: the number of interpolation steps will be chosen close to c1.B, where B is the bit accuracy.

```
? \p2000
? sumnumlagrange(n=1, n^-2);
time = 173 ms.
? tab = sumnumlagrangeinit();
time = 172 ms.
? sumnumlagrange(n=1, n^-2, tab);
time = 4 ms.
? \p115
? sumnumlagrange(n=1, n^(-4/3)) - zeta(4/3);
```

```
%1 = -0.1093[...] \\ junk: expansion in n^(1/3)
time = 84 ms.
? tab = sumnumlagrangeinit([1/3,0]); \\ alpha = 1/3
time = 336 ms.
? sumnumlagrange(n=1, n^(-4/3), tab) - zeta(4/3)
time = 84 ms.
%3 = 1.0151767349262596893 E-115 \\ now OK
? tab = sumnumlagrangeinit(1/3); \\ alpha = 1, beta = 1/3: much faster
time = 3ms
? sumnumlagrange(n=1, n^(-4/3), tab) - zeta(4/3) \\ ... but wrong
%5 = -0.273825[...] \\ junk !
? tab = sumnumlagrangeinit(-2/3); \\ alpha = 1, beta = -2/3
time = 3ms
? sumnumlagrange(n=1, n^(-4/3), tab) - zeta(4/3)
%6 = 2.030353469852519379 E-115 \\ now OK
```

in The final example with $\zeta(4/3)$, the remainder $R_1(n)$ is of the form $n^{-1/3} \sum_{i>=0} a_i/n^i$, i.e. $n^{2/3} \sum_{i>=1} a_i/n^i$. The explains the wrong result for $\beta = 1/3$ and the correction with $\beta = -2/3$.

sumnummonieninit(asymp, w, n0, precision)

Initialize tables for Monien summation of a series $\sum_{n>=n_0} f(n)$ where f(1/z) has a complex analytic continuation in a (complex) neighbourhood of the segment [0, 1].

By default, assume that $f(n) = O(n^{-2})$ and has a nonzero asymptotic expansion

$$f(n) = \sum_{i>=2} a_i/n^2$$

at infinity. Note that the sum starts at i = 2! The argument asymp allows to specify different expansions:

• a real number $\beta > 0$ means

$$f(n) = \sum_{i>=1} a_i / n^{i+\beta}$$
(Now
the summation starts at : math : '1'.)

• a vector $[\alpha, \beta]$ of reals, where we must have $\alpha > 0$ and $\alpha + \beta > 1$ to ensure convergence, means that

$$f(n) = \sum_{i>=1} a_i / n^{\alpha i + \beta}$$

 $Note that: math: `asymp = [1, \beta]` is equivalent to: math: `asymp = \beta`.$

```
? \p57
? s = sumnum(n = 1, sin(1/sqrt(n)) / n); \\ reference point
? \p38
? sumnummonien(n = 1, sin(1/sqrt(n)) / n) - s
%2 = -0.001[...] \\ completely wrong
? t = sumnummonieninit(1/2); \\ f(n) = sum_i 1 / n^(i+1/2)
? sumnummonien(n = 1, sin(1/sqrt(n)) / n, t) - s
%3 = 0.E-37 \\ now correct
```

(As a matter of fact, in the above summation, the result given by summum at p38 is slighly incorrect, so we had to increase the accuracy to p57.)

The argument w is used to sum expressions of the form

$$\sum_{n>=n_0} f(n)w(n)$$

for varying f as above, and fixed weight function w, where we further assume that the auxiliary sums

$$g_w(m) = \sum_{n > = n_0} w(n) / n^{\alpha m + \beta}$$

converge for all $m \ge 1$. Note that for nonnegative integers k, and weight $w(n) = (\log n)^k$, the function $g_w(m) = \zeta^{(k)}(\alpha m + \beta)$ has a simple expression; for general weights, g_w is computed using summum. The following variants are available

- an integer $k \ge 0$, to code $w(n) = (\log n)^k$;
- a t_CLOSURE computing the values w(n), where we assume that $w(n) = O(n^{\epsilon})$ for all $\epsilon > 0$;
- a vector [w, fast], where w is a closure as above and fast is a scalar; we assume that w(n) = O(n^{fast+ε}); note that w = [w, 0] is equivalent to w = w. Note that if w decreases exponentially, suminf should be used instead.

The subsequent calls to sumnummonien *must* use the same value of n_0 as was used here.

```
? \p300
? sumnummonien(n = 1, n^-2*log(n)) + zeta'(2)
time = 328 ms.
%1 = -1.323[...]E-6 \\ completely wrong, f does not satisfy hypotheses !
? tab = sumnummonieninit(, 1); \\ codes w(n) = log(n)
time = 3,993 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 41 ms.
%3 = -5.562684646268003458 E-309 \\ now perfect
? tab = sumnummonieninit(, n->log(n)); \\ generic, slower
time = 9,808 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 40 ms.
%5 = -5.562684646268003458 E-309 \\ identical result
```

sumnumrat(F, a, precision)

 $\sum_{n>=a} F(n)$, where F is a rational function of degree less than or equal to -2 and where poles of F at integers >=a are omitted from the summation. The argument a must be a t_INT or -oo.

? sumnumrat(1/(x²+1)²,0) %1 = 1.3068369754229086939178621382829073480 ? sumnumrat(1/x², -oo) \\ value at x=0 is discarded %2 = 3.2898681336964528729448303332920503784 ? 2*zeta(2) %3 = 3.2898681336964528729448303332920503784

When deg F = -1, we define

$$\sum_{-oo}^{oo} F(n) := \sum_{n \ge 0} (F(n) + F(-1 - n)) :$$

? sumnumrat(1/x, -oo) %4 = 0.E-38

system(str)

str is a string representing a system command. This command is executed, its output written to the standard output (this won't get into your logfile), and control returns to the PARI system. This simply calls the C system command.

tan(*x*, *precision*)

Tangent of x.

tanh(*x*, *precision*)

Hyperbolic tangent of x.

taylor(x, t, serprec)

Taylor expansion around 0 of x with respect to the simple variable t. x can be of any reasonable type, for example a rational function. Contrary to Ser, which takes the valuation into account, this function adds $O(t^d)$ to all components of x.

? taylor(x/(1+y), y, 5) %1 = (y^4 - y^3 + y^2 - y + 1)*x + 0(y^5) ? Ser(x/(1+y), y, 5) *** at top-level: Ser(x/(1+y),y,5) *** ^------*** Ser: main variable must have higher priority in gtoser.

teichmuller(x, tab)

Teichmüller character of the *p*-adic number *x*, i.e. the unique (p-1)-th root of unity congruent to $x/p^{v_p(x)}$ modulo *p*. If *x* is of the form [p, n], for a prime *p* and integer *n*, return the lifts to \mathbb{Z} of the images of $i + O(p^n)$ for i = 1, ..., p - 1, i.e. all roots of 1 ordered by residue class modulo *p*. Such a vector can be fed back to teichmuller, as the optional argument tab, to speed up later computations.

```
? z = teichmuller(2 + 0(101^5))
%1 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
? z^100
%2 = 1 + 0(101^5)
? T = teichmuller([101, 5]);
? teichmuller(2 + 0(101^5), T)
%4 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
```

As a rule of thumb, if more than

 $p/2(\log_2(p) + hammingweight(p))$

values of teichmuller are to be computed, then it is worthwile to initialize:

```
? p = 101; n = 100; T = teichmuller([p,n]); \\ instantaneous
? for(i=1,10^3, vector(p-1, i, teichmuller(i+0(p^n), T)))
time = 60 ms.
? for(i=1,10^3, vector(p-1, i, teichmuller(i+0(p^n))))
time = 1,293 ms.
? 1 + 2*(log(p)/log(2) + hammingweight(p))
%8 = 22.316[...]
```
Here the precomputation induces a speedup by a factor 1293/60 21.5.

Caveat. If the accuracy of tab (the argument n above) is lower than the precision of x, the *former* is used, i.e. the cached value is not refined to higher accuracy. It the accuracy of tab is larger, then the precision of x is used:

```
? Tlow = teichmuller([101, 2]); \\ lower accuracy !
? teichmuller(2 + 0(101^5), Tlow)
%10 = 2 + 83*101 + 0(101^5) \\ no longer a root of 1
? Thigh = teichmuller([101, 10]); \\ higher accuracy
? teichmuller(2 + 0(101^5), Thigh)
%12 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
```

theta(q, z, precision)

Jacobi sine theta-function

$$\theta_1(z,q) = 2q^{1/4} \sum_{n>=0} (-1)^n q^{n(n+1)} \sin((2n+1)z).$$

thetanullk(q, k, precision)

k-th derivative at z = 0 of theta(q, z).

thue(*tnf*, *a*, *sol*)

Returns all solutions of the equation P(x, y) = a in integers x and y, where *tnf* was created with *thueinit*(P). If present, *sol* must contain the solutions of Norm(x) = a modulo units of positive norm in the number field defined by P (as computed by bnfisintnorm). If there are infinitely many solutions, an error is issued.

It is allowed to input directly the polynomial P instead of a *tnf*, in which case, the function first performs thueinit(P, \emptyset). This is very wasteful if more than one value of a is required.

If *tnf* was computed without assuming GRH (flag 1 in thueinit), then the result is unconditional. Otherwise, it depends in principle of the truth of the GRH, but may still be unconditionally correct in some favorable cases. The result is conditional on the GRH if a! = 1 and P has a single irreducible rational factor, whose attached tentative class number h and regulator R (as computed assuming the GRH) satisfy

• *h* > 1,

•
$$R/0.2 > 1.5$$

Here's how to solve the Thue equation $x^{13} - 5y^{13} = -4$:

```
? tnf = thueinit(x^13 - 5);
? thue(tnf, -4)
%1 = [[1, 1]]
```

In this case, one checks that $bnfinit(x^13 -5)$.no is 1. Hence, the only solution is (x, y) = (1, 1) and the result is unconditional. On the other hand:

```
? P = x^3-2*x^2+3*x-17; tnf = thueinit(P);
? thue(tnf, -15)
%2 = [[1, 1]] \\ a priori conditional on the GRH.
? K = bnfinit(P); K.no
%3 = 3
? K.reg
%4 = 2.8682185139262873674706034475498755834
```

This time the result is conditional. All results computed using this particular *tnf* are likewise conditional, *except* for a right-hand side of 1. The above result is in fact correct, so we did not just disprove the GRH:

```
? tnf = thueinit(x^3-2*x^2+3*x-17, 1 /*unconditional*/);
? thue(tnf, -15)
%4 = [[1, 1]]
```

Note that reducible or nonmonic polynomials are allowed:

```
? tnf = thueinit((2*x+1)^5 * (4*x^3-2*x^2+3*x-17), 1);
? thue(tnf, 128)
%2 = [[-1, 0], [1, 0]]
```

Reducible polynomials are in fact much easier to handle.

Note. When P is irreducible without a real root, the default strategy is to use brute force enumeration in time $||a||^{1/\deg P}$ and avoid computing a tough *bnf* attached to P, see thueinit. Besides reusing a quantity you might need for other purposes, the default argument *sol* can also be used to use a different strategy and prove that there are no solutions; of course you need to compute a *bnf* on you own to obtain *sol*. If there *are* solutions this won't help unless P is quadratic, since the enumeration will be performed in any case.

thueinit(P, flag, precision)

Initializes the *tnf* corresponding to P, a nonconstant univariate polynomial with integer coefficients. The result is meant to be used in conjunction with thue to solve Thue equations $P(X/Y)Y^{\deg P} = a$, where a is an integer. Accordingly, P must either have at least two distinct irreducible factors over \mathbb{Q} , or have one irreducible factor T with degree > 2 or two conjugate complex roots: under these (necessary and sufficient) conditions, the equation has finitely many integer solutions.

```
? S = thueinit(t^2+1);
? thue(S, 5)
%2 = [[-2, -1], [-2, 1], [-1, -2], [-1, 2], [1, -2], [1, 2], [2, -1], [2, 1]]
? S = thueinit(t+1);
*** at top-level: thueinit(t+1)
*** ^------
*** thueinit: domain error in thueinit: P = t + 1
```

The hardest case is when $\deg P > 2$ and P is irreducible with at least one real root. The routine then uses Bilu-Hanrot's algorithm.

If flag is nonzero, certify results unconditionally. Otherwise, assume GRH, this being much faster of course. In the latter case, the result may still be unconditionally correct, see thue. For instance in most cases where P is reducible (not a pure power of an irreducible), *or* conditional computed class groups are trivial *or* the right hand side is 1, then results are unconditional.

Note. The general philosophy is to disprove the existence of large solutions then to enumerate bounded solutions naively. The implementation will overflow when there exist huge solutions and the equation has degree > 2 (the quadratic imaginary case is special, since we can stick to **bnfisintnorm**, there are no fundamental units):

```
? thue(t^3+2, 10^30)
  *** at top-level: L=thue(t^3+2,10^30)
  *** ^------
  *** thue: overflow in thue (SmallSols): y <= 80665203789619036028928.
? thue(x^2+2, 10^30) \\ quadratic case much easier
%1 = [[-10000000000000, 0], [1000000000000, 0]]</pre>
```

Note. It is sometimes possible to circumvent the above, and in any case obtain an important speed-up, if you can write $P = Q(x^d)$ for some d > 1 and Q still satisfying the thueinit hypotheses. You can then solve the equation attached to Q then eliminate all solutions (x, y) such that either x or y is not a d-th power.

```
? thue(x^4+1, 10^40); \\ stopped after 10 hours
? filter(L,d) =
my(x,y); [[x,y] | v<-L, ispower(v[1],d,&x)&&ispower(v[2],d,&y)];
? L = thue(x^2+1, 10^40);
? filter(L, 2)
%4 = [[0, 10000000000], [1000000000, 0]]
```

The last 2 commands use less than 20ms.

Note. When P is irreducible without a real root, the equation can be solved unconditionnally in time $||a||^{1/\deg P}$. When this latter quantity is huge and the equation has no solutions, this fact may still be ascertained via arithmetic conditions but this now implies solving norm equations, computing a *bnf* and possibly assuming the GRH. When there is no real root, the code does not compute a *bnf* (with certification if flag = 1) if it expects this to be an "easy" computation (because the result would only be used for huge values of a). See thue for a way to compute an expensive *bnf* on your own and still get a result where this default cheap strategy fails.

trace(x)

This applies to quite general x. If x is not a matrix, it is equal to the sum of x and its conjugate, except for polmods where it is the trace as an algebraic number.

For x a square matrix, it is the ordinary trace. If x is a nonsquare matrix (but not a vector), an error occurs.

truncate(x, e)

Truncates x and sets e to the number of error bits. When x is in \mathbb{R} , this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and the truncated value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. e > 0), the result is undefined and an error occurs if e was not given. The function applies componentwise on vector / matrices; e is then the maximal number of error bits. If x is a rational function, the result is the "integer part" (Euclidean quotient of numerator by denominator) and e is not set.

Note a very special use of truncate: when applied to a power series, it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a *p*-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

type(x)

This is useful only under gp. Returns the internal type name of the PARI object x as a string. Check out existing type names with the metacommand t. For example type(1) will return "t_INT".

unexportall()

Empty the list of variables exported to the parallel world.

valuation(x, p)

Computes the highest exponent of p dividing x. If p is of type integer, x must be an integer, an intmod whose modulus is divisible by p, a fraction, a q-adic number with q = p, or a polynomial or power series in which case the valuation is the minimum of the valuation of the coefficients.

If p is of type polynomial, x must be of type polynomial or rational function, and also a power series if x is a monomial. Finally, the valuation of a vector, complex or quadratic number is the minimum of the component valuations.

If x = 0, the result is +oo if x is an exact object. If x is a p-adic numbers or power series, the result is the exponent of the zero. Any other type combinations gives an error.

varhigher(name, v)

Return a variable *name* whose priority is higher than the priority of v (of all existing variables if v is omitted). This is a counterpart to varlower.

This routine is useful since new GP variables directly created by the interpreter always have lower priority than existing GP variables. When some basic objects already exist in a variable that is incompatible with some function requirement, you can now create a new variable with a suitable priority instead of changing variables in existing objects:

```
? K = nfinit(x^2+1);
? rnfequation(K,y^2-2)
*** at top-level: rnfequation(K,y^2-2)
*** ^______
*** rnfequation: incorrect priority in rnfequation: variable y >= x
? y = varhigher("y", x);
? rnfequation(K, y^2-2)
%3 = y^4 - 2*y^2 + 9
```

Caution 1. The *name* is an arbitrary character string, only used for display purposes and need not be related to the GP variable holding the result, nor to be a valid variable name. In particular the *name* can not be used to retrieve the variable, it is not even present in the parser's hash tables.

```
? x = varhigher("#");
? x^2
%2 = #^2
```

Caution 2. There are a limited number of variables and if no existing variable with the given display name has the requested priority, the call to varhigher uses up one such slot. Do not create new variables in this way unless it's absolutely necessary, reuse existing names instead and choose sensible priority requirements: if you only need a variable with higher priority than *x*, state so rather than creating a new variable with highest priority.

```
\\ quickly use up all variables
? n = 0; while(1,varhigher("tmp"); n++)
 *** at top-level: n=0;while(1,varhigher("tmp");n++)
 *** ^------
 *** varhigher: no more variables available.
 *** Break loop: type 'break' to go back to GP prompt
break> n
65510
 \\ infinite loop: here we reuse the same 'tmp'
? n = 0; while(1,varhigher("tmp", x); n++)
```

variable(x)

Gives the main variable of the object x (the variable with the highest priority used in x), and p if x is a p-adic number. Return 0 if x has no variable attached to it.

```
? variable(x^2 + y)
%1 = x
? variable(1 + 0(5^2))
```

%2 = 5
? variable([x,y,z,t])
%3 = x
? variable(1)
%4 = 0

The construction

if (!variable(x),...)

can be used to test whether a variable is attached to x.

If x is omitted, returns the list of user variables known to the interpreter, by order of decreasing priority. (Highest priority is initially x, which come first until varhigher is used.) If varhigher or varlower are used, it is quite possible to end up with different variables (with different priorities) printed in the same way: they will then appear multiple times in the output:

```
? varhigher("y");
? varlower("y");
? variable()
%4 = [y, x, y]
```

Using v = variable() then v[1], v[2], etc. allows to recover and use existing variables.

variables(x)

Returns the list of all variables occurring in object x (all user variables known to the interpreter if x is omitted), sorted by decreasing priority.

? variables([x^2 + y*z + O(t), a+x]) %1 = [x, y, z, t, a]

The construction

if (!variables(x),...)

can be used to test whether a variable is attached to x.

If varhigher or varlower are used, it is quite possible to end up with different variables (with different priorities) printed in the same way: they will then appear multiple times in the output:

```
? y1 = varhigher("y");
? y2 = varlower("y");
? variables(y*y1*y2)
%4 = [y, y, y]
```

varlower(name, v)

Return a variable *name* whose priority is lower than the priority of v (of all existing variables if v is omitted). This is a counterpart to varhigher.

New GP variables directly created by the interpreter always have lower priority than existing GP variables, but it is not easy to check whether an identifier is currently unused, so that the corresponding variable has the expected priority when it's created! Thus, depending on the session history, the same command may fail or succeed:

```
? t; z; \\ now t > z
? rnfequation(t^2+1,z^2-t)
```

```
*** at top-level: rnfequation(t^2+1,z^
*** ^_____
*** rnfequation: incorrect priority in rnfequation: variable t >= t
```

Restart and retry:

? z; t; \\ now z > t
? rnfequation(t^2+1,z^2-t)
%2 = z^4 + 1

It is quite annoying for package authors, when trying to define a base ring, to notice that the package may fail for some users depending on their session history. The safe way to do this is as follows:

```
? z; t; \\ In new session: now z > t
...
? t = varlower("t", 'z);
? rnfequation(t^2+1,z^2-2)
%2 = z^4 - 2*z^2 + 9
? variable()
%3 = [x, y, z, t]
```

```
? t; z; \\ In new session: now t > z
...
? t = varlower("t", 'z); \\ create a new variable, still printed "t"
? rnfequation(t^2+1,z^2-2)
%2 = z^4 - 2*z^2 + 9
? variable()
%3 = [x, y, t, z, t]
```

Now both constructions succeed. Note that in the first case, varlower is essentially a no-op, the existing variable t has correct priority. While in the second case, two different variables are displayed as t, one with higher priority than z (created in the first line) and another one with lower priority (created by varlower).

Caution 1. The *name* is an arbitrary character string, only used for display purposes and need not be related to the GP variable holding the result, nor to be a valid variable name. In particular the *name* can not be used to retrieve the variable, it is not even present in the parser's hash tables.

```
? x = varlower("#");
? x^2
%2 = #^2
```

Caution 2. There are a limited number of variables and if no existing variable with the given display name has the requested priority, the call to varlower uses up one such slot. Do not create new variables in this way unless it's absolutely necessary, reuse existing names instead and choose sensible priority requirements: if you only need a variable with higher priority than x, state so rather than creating a new variable with highest priority.

```
65510
\\ infinite loop: here we reuse the same 'tmp'
? n = 0; while(1,varlower("tmp", x); n++)
```

vecextract(x, y, z)

Extraction of components of the vector or matrix x according to y. In case x is a matrix, its components are the *columns* of x. The parameter y is a component specifier, which is either an integer, a string describing a range, or a vector.

If y is an integer, it is considered as a mask: the binary bits of y are read from right to left, but correspond to taking the components from left to right. For example, if $y = 13 = (1101)_2$ then the components 1,3 and 4 are extracted.

If y is a vector (t_VEC, t_COL or t_VECSMALL), which must have integer entries, these entries correspond to the component numbers to be extracted, in the order specified.

If y is a string, it can be

- a single (nonzero) index giving a component number (a negative index means we start counting from the end).
- a range of the form ":math:`a.:math:b", where :math:`a and b are indexes as above. Any of a and b can be omitted; in this case, we take as default values a = 1 and b = -1, i.e. the first and last components respectively. We then extract all components in the interval [a, b], in reverse order if b < a.

In addition, if the first character in the string is ^, the complement of the given set of indices is taken.

If z is not omitted, x must be a matrix. y is then the row specifier, and z the column specifier, where the component specifier is as explained above.

```
? v = [a, b, c, d, e];
? vecextract(v, 5) \\ mask
%1 = [a, c]
? vecextract(v, [4, 2, 1]) \\ component list
%2 = [d, b, a]
? vecextract(v, "2..4") \\ interval
%3 = [b, c, d]
? vecextract(v, "-1..-3") \\ interval + reverse order
%4 = [e, d, c]
? vecextract(v, "^2") \\ complement
%5 = [a, c, d, e]
? vecextract(matid(3), "2..", "..")
%6 =
[0 1 0]
[0 0 1]
```

The range notations v[i..j] and $v[^i]$ (for t_VEC or t_COL) and M[i..j, k..1] and friends (for t_MAT) implement a subset of the above, in a simpler and *faster* way, hence should be preferred in most common situations. The following features are not implemented in the range notation:

- · reverse order,
- omitting either *a* or *b* in :math:`a..:math:*b*`.

vecmax(x, v)

If x is a vector or a matrix, returns the largest entry of x, otherwise returns a copy of x. Error if x is empty.

If v is given, set it to the index of a largest entry (indirect maximum), when x is a vector. If x is a matrix, set v to coordinates [i, j] such that x[i, j] is a largest entry. This flag is ignored if x is not a vector or matrix.

```
? vecmax([10, 20, -30, 40])
%1 = 40
? vecmax([10, 20, -30, 40], &v); v
%2 = 4
? vecmax([10, 20; -30, 40], &v); v
%3 = [2, 2]
```

vecmin(x, v)

If x is a vector or a matrix, returns the smallest entry of x, otherwise returns a copy of x. Error if x is empty.

If v is given, set it to the index of a smallest entry (indirect minimum), when x is a vector. If x is a matrix, set v to coordinates [i, j] such that x[i, j] is a smallest entry. This is ignored if x is not a vector or matrix.

? vecmin([10, 20, -30, 40])
%1 = -30
? vecmin([10, 20, -30, 40], &v); v
%2 = 3
? vecmin([10, 20; -30, 40], &v); v
%3 = [2, 1]

vecprod(v)

Return the product of the components of the vector v. Return 1 on an empty vector.

```
? vecprod([1,2,3])
%1 = 6
? vecprod([])
%2 = 1
```

vecsearch(v, x, cmpf)

Determines whether x belongs to the sorted vector or list v: return the (positive) index where x was found, or 0 if it does not belong to v.

If the comparison function cmpf is omitted, we assume that v is sorted in increasing order, according to the standard comparison function lex, thereby restricting the possible types for x and the elements of v (integers, fractions, reals, and vectors of such). We also transparently allow a t_VECSMALL x in this case, for the natural ordering of the integers.

If cmpf is present, it is understood as a comparison function and we assume that v is sorted according to it, see vecsort for how to encode comparison functions.

```
? v = [1,3,4,5,7];
? vecsearch(v, 3)
%2 = 2
? vecsearch(v, 6)
%3 = 0 \\ not in the list
? vecsearch([7,6,5], 5) \\ unsorted vector: result undefined
%4 = 0
```

Note that if we are sorting with respect to a key which is expensive to compute (e.g. a discriminant), one should rather precompute all keys, sort that vector and search in the vector of keys, rather than searching in the original vector with respect to a comparison function.

By abuse of notation, x is also allowed to be a matrix, seen as a vector of its columns; again by abuse of notation, a t_VEC is considered as part of the matrix, if its transpose is one of the matrix columns.

```
? v = vecsort([3,0,2; 1,0,2]) \\ sort matrix columns according to lex order
%1 =
[0 2 3]
[0 2 1]
? vecsearch(v, [3,1]~)
%2 = 3
? vecsearch(v, [3,1]) \\ can search for x or x~
%3 = 3
? vecsearch(v, [1,2])
%4 = 0 \\ not in the list
```

```
vecsort(x, cmpf, flag)
```

Sorts the vector x in ascending order, using a mergesort method. x must be a list, vector or matrix (seen as a vector of its columns). Note that mergesort is stable, hence the initial ordering of "equal" entries (with respect to the sorting criterion) is not changed.

If cmpf is omitted, we use the standard comparison function lex, thereby restricting the possible types for the elements of x (integers, fractions or reals and vectors of those). We also transparently allow a t_VECSMALL x in this case, for the standard ordering on the integers.

If cmpf is present, it is understood as a comparison function and we sort according to it. The following possibilities exist:

- an integer k: sort according to the value of the k-th subcomponents of the components of x.
- a vector: sort lexicographically according to the components listed in the vector. For example, if cmpf = [2, 1, 3], sort with respect to the second component, and when these are equal, with respect to the first, and when these are equal, with respect to the third.
- a comparison function: t_CLOSURE with two arguments x and y, and returning a real number which is < 0,
 > 0 or = 0 if x < y, x > y or x = y respectively.
- a key: t_CLOSURE with one argument x and returning the value f(x) with respect to which we sort.

```
? vecsort([3,0,2; 1,0,2]) \\ sort columns according to lex order
%1 =
[0 2 3]
[0 2 1]
? vecsort(v, (x,y)->y-x) \\ reverse sort
? vecsort(v, (x,y)->abs(x)-abs(y)) \\ sort by increasing absolute value
? vecsort(v, abs) \\ sort by increasing absolute value, using key
? cmpf(x,y) = my(dx = poldisc(x), dy = poldisc(y)); abs(dx) - abs(dy);
? v = [x^2+1, x^3-2, x^4+5*x+1] vecsort(v, cmpf) \\ comparison function
? vecsort(v, x->abs(poldisc(x))) \\ key
```

The abs and cmpf examples show how to use a named function instead of an anonymous function. It is preferable to use a *key* whenever possible rather than include it in the comparison function as above since the key is evaluated O(n) times instead of $O(n \log n)$, where n is the number of entries.

A direct approach is also possible and equivalent to using a sorting key:

```
? T = [abs(poldisc(x)) | x<-v];
? perm = vecsort(T,,1); \\ indirect sort
? vecextract(v, perm)
```

This also provides the vector T of all keys, which is interesting for instance in later vecsearch calls: it is more efficient to sort T (T = vecextract(T, perm)) then search for a key in T rather than to search in v using a comparison function or a key. Note also that mapisdefined is often easier to use and faster than vecsearch.

The binary digits of *flag* mean:

- 1: indirect sorting of the vector x, i.e. if x is an n-component vector, returns a permutation of [1, 2, ..., n] which applied to the components of x sorts x in increasing order. For example, vecextract(x, vecsort(x,,1)) is equivalent to vecsort(x).
- 4: use descending instead of ascending order.
- 8: remove "duplicate" entries with respect to the sorting function (keep the first occurring entry). For example:

```
? vecsort([Pi,Mod(1,2),z], (x,y)->0, 8) \\ make everything compare equal
%1 = [3.141592653589793238462643383]
? vecsort([[2,3],[0,1],[0,3]], 2, 8)
%2 = [[0, 1], [2, 3]]
```

vecsum(v)

Return the sum of the components of the vector v. Return 0 on an empty vector.

```
? vecsum([1,2,3])
%1 = 6
? vecsum([])
%2 = 0
```

version()

Returns the current version number as a t_VEC with three integer components (major version number, minor version number and patchlevel); if your sources were obtained through our version control system, this will be followed by further more precise arguments, including e.g. a git *commit hash*.

This function is present in all versions of PARI following releases 2.3.4 (stable) and 2.4.3 (testing).

Unless you are working with multiple development versions, you probably only care about the 3 first numeric components. In any case, the lex function offers a clever way to check against a particular version number, since it will compare each successive vector entry, numerically or as strings, and will not mind if the vectors it compares have different lengths:

```
if (lex(version(), [2,3,5]) >= 0,
\\ code to be executed if we are running 2.3.5 or more recent.
,
\\ compatibility code
);
```

On a number of different machines, version() could return either of

```
%1 = [2, 3, 4] \\ released version, stable branch
%1 = [2, 4, 3] \\ released version, testing branch
%1 = [2, 6, 1, 15174, ""505ab9b"] \\ development
```

In particular, if you are only working with released versions, the first line of the gp introductory message can be emulated by

[M,m,p] = version(); printf("GP/PARI CALCULATOR Version %s.%s.%s", M,m,p);

If you *are* working with many development versions of PARI/GP, the 4th and/or 5th components can be profitably included in the name of your logfiles, for instance.

Technical note. For development versions obtained via git, the 4th and 5th components are liable to change eventually, but we document their current meaning for completeness. The 4th component counts the number of reachable commits in the branch (analogous to svn's revision number), and the 5th is the git commit hash. In particular, lex comparison still orders correctly development versions with respect to each others or to released versions (provided we stay within a given branch, e.g. master)!

weber(x, flag, precision)

One of Weber's three f functions. If flag = 0, returns

$$f(x) = \exp(-i\pi/24) \cdot \eta((x+1)/2)/\eta(x) such that j = (f^{24} - 16)^3/f^{24}$$

where j is the elliptic j-invariant (see the function ellj). If flag = 1, returns

 $f_1(x) = \eta(x/2)/\eta(x)such that j = (f_1^{24} + 16)^3/f_1^{24}.$

Finally, if flag = 2, returns

$$f_2(x) = \sqrt{2\eta(2x)}/\eta(x) such that j = (f_2^{24} + 16)^3/f_2^{24}.$$

Note the identities $f^8 = f_1^8 + f_2^8$ and $ff_1f_2 = \sqrt{2}$.

writebin(filename, x)

Writes (appends) to *filename* the object x in binary format. This format is not human readable, but contains the exact internal structure of x, and is much faster to save/load than a string expression, as would be produced by write. The binary file format includes a magic number, so that such a file can be recognized and correctly input by the regular read or r function. If saved objects refer to polynomial variables that are not defined in the new session, they will be displayed as t:math: `n` for some integer n (the attached variable number). Installed functions and history objects can not be saved via this function.

If x is omitted, saves all user variables from the session, together with their names. Reading such a "named object" back in a gp session will set the corresponding user variable to the saved value. E.g after

x = 1; writebin("log")

reading log into a clean session will set x to 1. The relative variables priorities (see priority (in the PARI manual)) of new variables set in this way remain the same (preset variables retain their former priority, but are set to the new value). In particular, reading such a session log into a clean session will restore all variables exactly as they were in the original one.

Just as a regular input file, a binary file can be compressed using gzip, provided the file name has the standard .gz extension.

In the present implementation, the binary files are architecture dependent and compatibility with future versions of gp is not guaranteed. Hence binary files should not be used for long term storage (also, they are larger and harder to compress than text files).

zeta(*s*, *precision*)

For s! = 1 a complex number, Riemann's zeta function $\zeta(s) = \sum_{n>=1} n^{-s}$, computed using the Euler-Maclaurin summation formula, except when s is of type integer, in which case it is computed using Bernoulli numbers for s <= 0 or s > 0 and even, and using modular forms for s > 0 and odd. Power series are also allowed:

For s! = 1 a *p*-adic number, Kubota-Leopoldt zeta function at *s*, that is the unique continuous *p*-adic function on the *p*-adic integers that interpolates the values of $(1 - p^{-k})\zeta(k)$ at negative integers *k* such that k = 1(modp - 1) (resp. *k* is odd) if *p* is odd (resp. p = 2). Power series are not allowed in this case.

zetahurwitz(s, x, der, precision)

Hurwitz zeta function $\zeta(s, x) = \sum_{n>=0} (n+x)^{-s}$ and analytically continued, with s! = 1 and x not a negative or zero integer. Note that $\zeta(s, 1) = \zeta(s)$. s can also be a polynomial, rational function, or power series. If der is positive, compute the der'th derivative with respect to s. Note that the derivative with respect to x is simply $-s\zeta(s+1, x)$.

```
? zetahurwitz(Pi,Pi)
%1 = 0.056155444497585099925180502385781494484
? zetahurwitz(2,1) - zeta(2)
%2 = -2.350988701644575016 E-38
? zetahurwitz(Pi,3) - (zeta(Pi)-1-1/2^Pi)
%3 = -2.2040519077917890774 E-39
? zetahurwitz(-7/2,1) - zeta(-7/2)
\%4 = -2.295887403949780289 E-41
? zetahurwitz(-2.3,Pi+I*log(2))
\%5 = -5.1928369229555125820137832704455696057
- 6.1349660138824147237884128986232049582*I
? zetahurwitz(-1+x^2+0(x^3),1)
- 0.16542114370045092921391966024278064276*x^2 + 0(x^3)
? zetahurwitz(1+x+0(x^4),2)
- 0.42278433509846713939348790991759756896
+ 0.072815845483676724860586375874901319138*x + 0(x^2)
? zetahurwitz(2,1,2) \\ zeta''(2)
%8 = 1.9892802342989010234208586874215163815
```

zetamult(s, t, precision)

For s a vector of positive integers such that $s[1] \ge 2$, returns the multiple zeta value (MZV)

$$\zeta(s_1, ..., s_k) = \sum_{n_1 > ... > n_k > 0} n_1^{-s_1} ... n_k^{-s_k}$$

of length k and weight $\sum_i s_i$. More generally, return Yamamoto's t-MZV interpolation evaluated at t: for t = 0, this is the ordinary MZV; for t = 1, we obtain the MZSV star value, with >= instead of strict inequalities; and of course, for t = x we obtain Yamamoto's one-variable polynomial.

```
? zetamult([2,1]) - zeta(3) \\ Euler's identity
%1 = 0.E-38
? zetamult([2,1], 1) \\ star value
%2 = 2.4041138063191885707994763230228999815
? zetamult([2,1], 'x)
%3 = 1.20205[...]*x + 1.20205[...]
```

If the bit precision is B, this function runs in time $O(k(B+k)^2)$ if t = 0, and $O(kB^3)$ otherwise.

In addition to the above format (avec), the function also accepts a binary word format evec (each s_i is replaced by s_i bits, all of them 0 but the last one) giving the MZV representation as an iterated integral, and an index format (if e is the positive integer attached the evec vector of bits, the index is the integer $e + 2^{k-2}$). The function zetamultconvert allows to pass from one format to the other; the function zetamultall computes simultaneously all MZVs of weight $\sum_{i < =k} s_i$ up to n.

zetamultall(k, flag, precision)

List of all multiple zeta values (MZVs) for weight $s_1 + ... + s_r$ up to k. Binary digits of flag mean: 0 =star values if set; 1 = values up to to duality if set (see zetamultdual, ignored if star values); 2 = values of weight k if set (else all values up to weight k); 3 = return the 2-component vector [Z, M], where M is the vector of the corresponding indices m, i.e., such that zetamult(M[i]) = Z[i]. Note that it is necessary to use zetamultconvert to have the corresponding avec $(s_1, ..., s_r)$.

With default flag flag = 0, the function returns a vector with $2^{k-1} - 1$ components whose *i*-th entry is the MZV of index *i* (see zetamult). If the bit precision is *B*, this function runs in time $O(2^k k B^2)$ for an output of size $O(2^k B)$.

```
? Z = zetamultall(5); #Z \\ 2^4 - 1 MZVs of weight <= 5
%1 = 15
? Z[10]
%2 = 0.22881039760335375976874614894168879193
? zetamultconvert(10)
%3 = Vecsmall([3, 2]) \\ {index 10 corresponds to zeta (3,2)}
? zetamult(%) \\ double check
%4 = 0.22881039760335375976874614894168879193
? zetamult(10) \\ we can use the index directly
%5 = 0.22881039760335375976874614894168879193
```

If we use flag bits 1 and 2, we avoid unnecessary computations and copying, saving a potential factor 4: half the values are in lower weight and computing up to duality save another rough factor 2. Unfortunately, the indexing now no longer corresponds to the new shorter vector of MZVs:

```
? Z = zetamultall(5, 2); #Z \\ up to duality
%6 = 9
? Z = zetamultall(5, 2); #Z \\ only weight 5
%7 = 8
? Z = zetamultall(5, 2 + 4); #Z \\ both
%8 = 4
```

So how to recover the value attached to index 10 ? Flag bit 3 returns the actual indices used:

```
? [Z, M] = zetamultall(5, 2 + 8); M \\ other indices were not included
%9 = Vecsmall([1, 2, 4, 5, 6, 8, 9, 10, 12])
? Z[8] \\ index m = 10 is now in M[8]
%10 = 0.22881039760335375976874614894168879193
```

```
? [Z, M] = zetamultall(5, 2 + 4 + 8); M
%11 = Vecsmall([8, 9, 10, 12])
? Z[3] \\ index m = 10 is now in M[3]
%12 = 0.22881039760335375976874614894168879193
```

The following construction automates the above programmatically, looking up the MZVs of index $10 (= \zeta(3, 2))$ in all cases, without inspecting the various index sets M visually:

? Z[vecsearch(M, 10)] \\ works in all the above settings %13 = 0.22881039760335375976874614894168879193

zetamultconvert(a, fl)

a being either an evec, avec, or index m, converts into evec (fl = 0), avec (fl = 1), or index m (fl = 2).

```
? zetamultconvert(10)
%1 = Vecsmall([3, 2])
? zetamultconvert(13)
%2 = Vecsmall([2, 2, 1])
? zetamultconvert(10, 0)
%3 = Vecsmall([0, 0, 1, 0, 1])
? zetamultconvert(13, 0)
%4 = Vecsmall([0, 1, 0, 1, 1])
```

The last two lines imply that [3, 2] and [2, 2, 1] are dual (reverse order of bits and swap 0 and 1 in evec form). Hence they have the same zeta value:

```
? zetamult([3,2])
%5 = 0.22881039760335375976874614894168879193
? zetamult([2,2,1])
%6 = 0.22881039760335375976874614894168879193
```

zetamultdual(s)

s being either an evec, avec, or index m, return the dual sequence in avec format. The dual of a sequence of length r and weight k has length k - r and weight k. Duality is an involution and zeta values attached to dual sequences are the same:

```
? zetamultdual([4])
%1 = Vecsmall([2, 1, 1])
? zetamultdual(%)
%2 = Vecsmall([4])
? zetamult(%1) - zetamult(%2)
%3 = 0.E-38
```

In evec form, duality simply reverses the order of bits and swaps 0 and 1:

```
? zetamultconvert([4], 0)
%4 = Vecsmall([0, 0, 0, 1])
? zetamultconvert([2,1,1], 0)
%5 = Vecsmall([0, 1, 1, 1])
```

znchar(D)

Given a datum D describing a group $(\mathbb{Z}/N\mathbb{Z})^*$ and a Dirichlet character χ , return the pair [G, chi], where G is znstar(N, 1)) and chi is a GP character.

The following possibilities for D are supported

- a nonzero t_INT congruent to 0, 1 modulo 4, return the real character modulo D given by the Kronecker symbol (D/.);
- a t_INTMOD Mod(m, N), return the Conrey character modulo N of index m (see znconreylog).
- a modular form space as per $mfinit([N, k, \chi])$ or a modular form for such a space, return the underlying Dirichlet character χ (which may be defined modulo a divisor of N but need not be primitive).

In the remaining cases, G is initialized by znstar(N, 1).

- a pair [G, chi], where chi is a standard GP Dirichlet character c = (c_j) on G (generic character t_VEC or Conrey characters t_COL or t_INT); given generators G = ⊕(ℤ/d_jℤ)g_j, χ(g_j) = e(c_j/d_j).
- a pair [G, chin], where chin is a *normalized* representation [n, c] of the Dirichlet character c; $\chi(g_j) = e(c_j/n)$ where n is minimal (order of χ).

```
? [G,chi] = znchar(-3);
? G.cyc
%2 = [2]
? chareval(G, chi, 2)
%3 = 1/2
? kronecker(-3,2)
%4 = -1
? znchartokronecker(G,chi)
%5 = -3
? mf = mfinit([28, 5/2, Mod(2,7)]); [f] = mfbasis(mf);
? [G,chi] = znchar(mf); [G.mod, chi]
%7 = [7, [2]~]
? [G,chi] = znchar(f); chi
%8 = [28, [0, 2]~]
```

zncharconductor(G, chi)

Let *G* be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$ (see dirichletchar (in the PARI manual) or ??character). Return the conductor of chi:

```
? G = znstar(126000, 1);
? zncharconductor(G,11) \\ primitive
%2 = 126000
? zncharconductor(G,1) \\ trivial character, not primitive!
%3 = 1
? zncharconductor(G,1009) \\ character mod 5^3
%4 = 125
```

znchardecompose(G, chi, Q)

Let $N = \prod_p p^{e_p}$ and a Dirichlet character χ , we have a decomposition $\chi = \prod_p \chi_p$ into character modulo N where the conductor of χ_p divides p^{e_p} ; it equals p^{e_p} for all p if and only if χ is primitive.

Given a *znstar* G describing a group $(\mathbb{Z}/N\mathbb{Z})^*$, a Dirichlet character chi and an integer Q, return $\prod_{p \parallel (Q,N)} \chi_p$. For instance, if Q = p is a prime divisor of N, the function returns χ_p (as a character modulo N), given as a Conrey character (t_COL).

```
? G = znstar(40, 1);
? G.cyc
%2 = [4, 2, 2]
? chi = [2, 1, 1];
```

```
? chi2 = znchardecompose(G, chi, 2)
%4 = [1, 1, 0]~
? chi5 = znchardecompose(G, chi, 5)
%5 = [0, 0, 2]~
? znchardecompose(G, chi, 3)
%6 = [0, 0, 0]~
? c = charmul(G, chi2, chi5)
%7 = [1, 1, 2]~ \\ t_COL: in terms of Conrey generators !
? znconreychar(G,c)
%8 = [2, 1, 1] \\ t_VEC: in terms of SNF generators
```

znchargauss(G, chi, a, precision)

Given a Dirichlet character χ on $G = (\mathbb{Z}/N\mathbb{Z})^*$ (see znchar), return the complex Gauss sum

$$g(\chi, a) = \sum_{n=1}^{N} \chi(n) e(an/N)$$

```
? [G,chi] = znchar(-3); \\ quadratic Gauss sum: I*sqrt(3)
? znchargauss(G,chi)
%2 = 1.7320508075688772935274463415058723670*I
? [G,chi] = znchar(5);
? znchargauss(G,chi) \\ sqrt(5)
%2 = 2.2360679774997896964091736687312762354
? G = znstar(300,1); chi = [1,1,12]~;
? znchargauss(G,chi) / sqrt(300) - exp(2*I*Pi*11/25) \\ = 0
%4 = 2.350988701644575016 E-38 + 1.4693679385278593850 E-39*I
? lfuntheta([G,chi], 1) \\ = 0
%5 = -5.79[...] E-39 - 2.71[...] E-40*I
```

zncharinduce(G, chi, N)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Let N be a multiple of q, return the character modulo N extending chi. As usual for arithmetic functions, the new modulus N can be given as a t_INT, via a factorization matrix or a pair [N, factor(N)], or by znstar(N, 1).

```
? G = znstar(4, 1);
? chi = znconreylog(G,1); \\ trivial character mod 4
? zncharinduce(G, chi, 80) \\ now mod 80
%3 = [0, 0, 0]~
? zncharinduce(G, 1, 80) \\ same using directly Conrey label
%4 = [0, 0, 0]~
? G2 = znstar(80, 1);
? zncharinduce(G, 1, G2) \\ same
%4 = [0, 0, 0]~
? chi = zncharinduce(G, 3, G2) \\ extend the nontrivial character mod 4
```

```
%5 = [1, 0, 0]~
? [G0,chi0] = znchartoprimitive(G2, chi);
? G0.mod
%7 = 4
? chi0
%8 = [1]~
```

Here is a larger example:

```
? G = znstar(126000, 1);
? label = 1009;
? chi = znconreylog(G, label)
%3 = [0, 0, 0, 14, 0]~
? [G0,chi0] = znchartoprimitive(G, label); \\ works also with 'chi'
? G0.mod
%5 = 125
? chi0 \\ primitive character mod 5^3 attached to chi
%6 = [14]~
? G0 = znstar(N0, 1);
? zncharinduce(G0, chi0, G) \\ induce back
%8 = [0, 0, 0, 14, 0]~
? znconreyexp(G, %)
%9 = 1009
```

zncharisodd(G, chi)

Let G be attached to $(\mathbb{Z}/N\mathbb{Z})^*$ (as per G = znstar(N,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/N\mathbb{Z})^*$, given by

- a t_VEC: a standard character on G.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Return 1 if and only if chi(-1) = -1 and 0 otherwise.

```
? G = znstar(8, 1);
? zncharisodd(G, 1) \\ trivial character
%2 = 0
? zncharisodd(G, 3)
%3 = 1
? chareval(G, 3, -1)
%4 = 1/2
```

znchartokronecker(G, chi, flag)

Let G be attached to $(\mathbb{Z}/N\mathbb{Z})^*$ (as per G = znstar(N,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/N\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

If flag = 0, return the discriminant D if chi is real equal to the Kronecker symbol (D/.) and 0 otherwise. The discriminant D is fundamental if and only if chi is primitive.

If flag = 1, return the fundamental discriminant attached to the corresponding primitive character.

? G = znstar(8,1); CHARS = [1,3,5,7]; \\ Conrey labels ? apply(t->znchartokronecker(G,t), CHARS) %2 = [4, -8, 8, -4] ? apply(t->znchartokronecker(G,t,1), CHARS) %3 = [1, -8, 8, -4]

znchartoprimitive(G, chi)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, of conductor $q_0 || q$.

```
? G = znstar(126000, 1);
? [G0,chi0] = znchartoprimitive(G,11)
? G0.mod
%3 = 126000
? chi0
\%4 = 11
? [G0, chi0] = znchartoprimitive(G,1);\\ trivial character, not primitive!
? G0.mod
\%6 = 1
? chi0
%7 = []~
? [G0,chi0] = znchartoprimitive(G,1009)
? G0.mod
\%4 = 125
? chi0
%5 = [14]~
```

Note that znconreyconductor is more efficient since it can return χ_0 and its conductor q_0 without needing to initialize G_0 . The price to pay is a more cryptic format and the need to initialize G_0 later, but that needs to be done only once for all characters with conductor q_0 .

znconreychar(G, m)

Given a *znstar* G attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q,1)), this function returns the Dirichlet character attached to $m \in (\mathbb{Z}/q\mathbb{Z})^*$ via Conrey's logarithm, which establishes a "canonical" bijection between $(\mathbb{Z}/q\mathbb{Z})^*$ and its dual.

Let $q = \prod_p p^{e_p}$ be the factorization of q into distinct primes. For all odd p with $e_p > 0$, let g_p be the element in $(\mathbb{Z}/q\mathbb{Z})^*$ which is

- congruent to $1 \mod q/p^{e_p}$,
- congruent mod p^{e_p} to the smallest positive integer that generates $(\mathbb{Z}/p^2\mathbb{Z})^*$.

For p = 2, we let g_4 (if $2^{e_2} \ge 4$) and g_8 (if furthermore $(2^{e_2} \ge 8)$ be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to $1 \mod q/2^{e_2}$,
- $g_4 = -1 \mod 2^{e_2}$,
- $g_8 = 5mod2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} \ge 2$) are independent generators of $(\mathbb{Z}/q\mathbb{Z})^*$, i.e. every m in $(\mathbb{Z}/q\mathbb{Z})^*$ can be written uniquely as $\prod_p g_p^{m_p}$, where m_p is defined modulo the order o_p of g_p and $p \in S_q$, the set of prime divisors of q together with 4 if 4 || q and 8 if 8 || q. Note that the g_p are in general *not* SNF generators as produced by znstar whenever $\omega(q) \ge 2$, although their number is the same. They however allow to handle the finite abelian group $(\mathbb{Z}/q\mathbb{Z})^*$ in a fast and elegant way. (Which unfortunately does not generalize to ray class groups or Hecke characters.) The Conrey logarithm of m is the vector $(m_p)_{p \in S_q}$, obtained via znconreylog. The Conrey character $\chi_q(m, .)$ attached to $m \mod q$ maps each g_p , $p \in S_q$ to $e(m_p/o_p)$, where $e(x) = \exp(2i\pi x)$. This function returns the Conrey character expressed in the standard PARI way in terms of the SNF generators G.gen.

```
? G = znstar(8,1);
? G.cyc
\%2 = [2, 2] \setminus Z/2 \times Z/2
? G.gen
\%3 = [7, 3]
? znconreychar(G,1) \setminus 1 is always the trivial character
\%4 = [0, 0]
? znconreychar(G,2) \\ 2 is not coprime to 8 !!!
*** at top-level: znconreychar(G,2)
*** ^_____
 *** znconreychar: elements not coprime in Zideallog:
2
8
*** Break loop: type 'break' to go back to GP prompt
break>
? znconreychar(G,3)
\%5 = [0, 1]
? znconreychar(G,5)
\%6 = [1, 1]
? znconreychar(G,7)
\%7 = [1, 0]
```

We indeed get all 4 characters of $(\mathbb{Z}/8\mathbb{Z})^*$.

For convenience, we allow to input the Conrey logarithm of m instead of m:

```
? G = znstar(55, 1);
? znconreychar(G,7)
%2 = [7, 0]
? znconreychar(G, znconreylog(G,7))
%3 = [7, 0]
```

znconreyconductor(*G*, *chi*, *chi0*)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in $(\mathbb{Z}/q\mathbb{Z})^*$ or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Return the conductor of chi, as the t_INT bid.mod if chi is primitive, and as a pair [N, faN] (with faN the factorization of N) otherwise.

If chi0 is present, set it to the Conrey logarithm of the attached primitive character.

```
? G = znstar(126000, 1);
? znconreyconductor(G,11) \\ primitive
%2 = 126000
? znconreyconductor(G,1) \\ trivial character, not primitive!
%3 = [1, matrix(0,2)]
```

```
? N0 = znconreyconductor(G,1009, &chi0) \\ character mod 5^3
%4 = [125, Mat([5, 3])]
? chi0
%5 = [14]~
? G0 = znstar(N0, 1); \\ format [N,factor(N)] accepted
? znconreyexp(G0, chi0)
%7 = 9
? znconreyconductor(G0, chi0) \\ now primitive, as expected
%8 = 125
```

The group GO is not computed as part of znconreyconductor because it needs to be computed only once per conductor, not once per character.

znconreyexp(G, chi)

Given a *znstar* G attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), this function returns the Conrey exponential of the character *chi*: it returns the integer $m \in (\mathbb{Z}/q\mathbb{Z})^*$ such that znconreylog(G, :math:`m)` is *chi*.

The character chi is given either as a

- t_VEC: in terms of the generators G.gen;
- t_COL: a Conrey logarithm.

```
? G = znstar(126000, 1)
? znconreylog(G,1)
\%2 = [0, 0, 0, 0, 0] \sim
? znconreyexp(G,%)
\%3 = 1
? G.cyc \\ SNF generators
\%4 = [300, 12, 2, 2, 2]
? chi = [100, 1, 0, 1, 0]; \\ some random character on SNF generators
? znconreylog(G, chi) \\ in terms of Conrey generators
\%6 = [0, 3, 3, 0, 2] \sim
? znconreyexp(G, \%) \setminus apply to a Conrey log
\%7 = 18251
? znconreyexp(G, chi) \\ ... or a char on SNF generators
\%8 = 18251
? znconreychar(G,%)
\%9 = [100, 1, 0, 1, 0]
```

znconreylog(G, m)

Given a *znstar* attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), this function returns the Conrey logarithm of $m \in (\mathbb{Z}/q\mathbb{Z})^*$.

Let $q = \prod_p p^{e_p}$ be the factorization of q into distinct primes, where we assume $e_2 = 0$ or $e_2 \ge 2$. (If $e_2 = 1$, we can ignore 2 from the factorization, as if we replaced q by q/2, since $(\mathbb{Z}/q\mathbb{Z})^*$ $(\mathbb{Z}/(q/2)\mathbb{Z})^*$.)

For all odd p with $e_p > 0$, let g_p be the element in $(\mathbb{Z}/q\mathbb{Z})^*$ which is

- congruent to $1 \mod q/p^{e_p}$,
- congruent mod p^{e_p} to the smallest positive integer that generates $(\mathbb{Z}/p^2\mathbb{Z})^*$.

For p = 2, we let g_4 (if $2^{e_2} \ge 4$) and g_8 (if furthermore $(2^{e_2} \ge 8)$ be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to $1 \mod q/2^{e_2}$,
- $g_4 = -1 \mod 2^{e_2}$,

• $g_8 = 5mod2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} \ge 2$) are independent generators of $\mathbb{Z}/q\mathbb{Z}^*$, i.e. every m in $(\mathbb{Z}/q\mathbb{Z})^*$ can be written uniquely as $\prod_p g_p^{m_p}$, where m_p is defined modulo the order o_p of g_p and $p \in S_q$, the set of prime divisors of q together with 4 if 4 || q and 8 if 8 || q. Note that the g_p are in general *not* SNF generators as produced by znstar whenever $\omega(q) \ge 2$, although their number is the same. They however allow to handle the finite abelian group $(\mathbb{Z}/q\mathbb{Z})^*$ in a fast and elegant way. (Which unfortunately does not generalize to ray class groups or Hecke characters.)

The Conrey logarithm of m is the vector $(m_p)_{p \in S_q}$. The inverse function znconreyexp recovers the Conrey label m from a character.

```
? G = znstar(126000, 1);
? znconreylog(G,1)
%2 = [0, 0, 0, 0, 0]~
? znconreyexp(G, %)
\%3 = 1
? znconreylog(G,2) \setminus 2 is not coprime to modulus !!!
*** at top-level: znconreylog(G,2)
 *** znconreylog: elements not coprime in Zideallog:
 2
 126000
 *** Break loop: type 'break' to go back to GP prompt
break>
? znconreylog(G,11) \\ wrt. Conrey generators
\%4 = [0, 3, 1, 76, 4] \sim
? log11 = ideallog(,11,G) \\ wrt. SNF generators
\%5 = [178, 3, -75, 1, 0] \sim
```

For convenience, we allow to input the ordinary discrete log of m, ideallog(, m, bid), which allows to convert discrete logs from bid.gen generators to Conrey generators.

? znconreylog(G, log11)
%7 = [0, 3, 1, 76, 4]~

We also allow a character (t_VEC) on bid.gen and return its representation on the Conrey generators.

```
? G.cyc
%8 = [300, 12, 2, 2, 2]
? chi = [10,1,0,1,1];
? znconreylog(G, chi)
%10 = [1, 3, 3, 10, 2]~
? n = znconreyexp(G, chi)
%11 = 84149
? znconreychar(G, n)
%12 = [10, 1, 0, 1, 1]
```

zncoppersmith(P, N, X, B)

Coppersmith's algorithm. N being an integer and $P \in \mathbb{Z}[t]$, finds in polynomial time in $\log(N)$ and d = deg(P) all integers x with $||x|| \le X$ such that

$$gcd(N, P(x)) >= B$$

This is a famous application of the LLL algorithm meant to help in the factorization of N. Notice that P may be reduced modulo $N\mathbb{Z}[t]$ without affecting the situation. The parameter X must not be too large: assume for now

that the leading coefficient of P is coprime to N, then we must have

$$d\log X\log N < \log^2 B,$$

i.e., $X < N^{1/d}$ when B = N. Let now P_0 be the gcd of the leading coefficient of P and N. In applications to factorization, we should have $P_0 = 1$; otherwise, either $P_0 = N$ and we can reduce the degree of P, or P_0 is a non trivial factor of N. For completeness, we nevertheless document the exact conditions that X must satisfy in this case: let $p := \log_N P_0$, $b := \log_N B$, $x := \log_N X$, then

- either $p \ge d/(2d-1)$ is large and we must have xd < 2b-1;
- or p < d/(2d-1) and we must have both p < b < 1 p + p/d and $x(d + p(1 2d)) < (b p)^2$. Note that this reduces to $xd < b^2$ when p = 0, i.e., the condition described above.

Some x larger than X may be returned if you are very lucky. The routine runs in polynomial time in $\log N$ and d but the smaller B, or the larger X, the slower. The strength of Coppersmith method is the ability to find roots modulo a general *composite* N: if N is a prime or a prime power, polrootsmod or polrootspadic will be much faster.

We shall now present two simple applications. The first one is finding nontrivial factors of N, given some partial information on the factors; in that case B must obviously be smaller than the largest nontrivial divisor of N.

```
setrand(1); \\ to make the example reproducible
[a,b] = [10^30, 10^31]; D = 20;
p = randomprime([a,b]);
q = randomprime([a,b]); N = p*q;
\\ assume we know 0) p | N; 1) p in [a,b]; 2) the last D digits of p
p0 = p % 10^D;
? L = zncoppersmith(10^D*x + p0, N, b \ 10^D, a)
time = 1ms.
%6 = [738281386540]
? gcd(L[1] * 10^D + p0, N) == p
%7 = 1
```

and we recovered p, faster than by trying all possibilities $x < 10^{11}$.

The second application is an attack on RSA with low exponent, when the message x is short and the padding P is known to the attacker. We use the same RSA modulus N as in the first example:

```
setrand(1);
P = random(N); \\ known padding
e = 3; \\ small public encryption exponent
X = floor(N^0.3); \\ N^(1/e - epsilon)
x0 = random(X); \\ unknown short message
C = lift( (Mod(x0,N) + P)^e); \\ known ciphertext, with padding P
zncoppersmith((P + x)^3 - C, N, X)
\\ result in 244ms.
%14 = [2679982004001230401]
? %[1] == x0
%15 = 1
```

We guessed an integer of the order of 10^{18} , almost instantly.

znlog(x, g, o)

This functions allows two distinct modes of operation depending on g:

- if g is the output of znstar (with initialization), we compute the discrete logarithm of x with respect to the generators contained in the structure. See ideallog for details.
- else g is an explicit element in (Z/NZ)*, we compute the discrete logarithm of x in (Z/NZ)* in base g. The
 rest of this entry describes the latter possibility.

The result is [] when x is not a power of g, though the function may also enter an infinite loop in this case.

If present, o represents the multiplicative order of g, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the order of g. This provides a definite speedup when the discrete log problem is simple:

```
? p = nextprime(10^4); g = znprimroot(p); o = [p-1, factor(p-1)];
? for(i=1,10^4, znlog(i, g, o))
time = 163 ms.
? for(i=1,10^4, znlog(i, g))
time = 200 ms. \\ a little slower
```

The result is undefined if g is not invertible mod N or if the supplied order is incorrect.

This function uses

- a combination of generic discrete log algorithms (see below).
- in (Z/NZ)^{*} when N is prime: a linear sieve index calculus method, suitable for N < 10⁵⁰, say, is used for large prime divisors of the order.

The generic discrete log algorithms are:

- Pohlig-Hellman algorithm, to reduce to groups of prime order q, where q || p 1 and p is an odd prime divisor of N,
- Shanks baby-step/giant-step ($q < 2^{32}$ is small),
- Pollard rho method $(q > 2^{32})$.

The latter two algorithms require $O(\sqrt{q})$ operations in the group on average, hence will not be able to treat cases where $q > 10^{30}$, say. In addition, Pollard rho is not able to handle the case where there are no solutions: it will enter an infinite loop.

```
? g = znprimroot(101)
\%1 = Mod(2, 101)
? znlog(5, g)
%2 = 24
? g^24
\%3 = Mod(5, 101)
? G = znprimroot(2 * 101^{10})
%4 = Mod(110462212541120451003, 220924425082240902002)
? znlog(5, G)
%5 = 76210072736547066624
? G^% == 5
\%6 = 1
? N = 2^4*3^2*5^3*7^4*11; g = Mod(13, N); znlog(g^110, g)
\%7 = 110
? znlog(6, Mod(2,3)) \setminus no solution
%8 = []
```

For convenience, g is also allowed to be a p-adic number:

? g = 3+0(5^10); znlog(2, g) %1 = 1015243 ? g^% %2 = 2 + 0(5^10)

znorder(x, o)

x must be an integer mod n, and the result is the order of x in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. Returns an error if x is not invertible. The parameter o, if present, represents a nonzero multiple of the order of x, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord = eulerphi(n) is the cardinality of the group.

znprimroot(n)

Returns a primitive root (generator) of $(\mathbb{Z}/n\mathbb{Z})^*$, whenever this latter group is cyclic (n = 4 or $n = 2p^k$ or $n = p^k$, where p is an odd prime and $k \ge 0$). If the group is not cyclic, the result is undefined. If n is a prime power, then the smallest positive primitive root is returned. This may not be true for $n = 2p^k$, p odd.

Note that this function requires factoring p-1 for p as above, in order to determine the exact order of elements in $(\mathbb{Z}/n\mathbb{Z})^*$: this is likely to be costly if p is large.

znstar(n, flag)

Gives the structure of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. The output G depends on the value of *flag*:

• flag = 0 (default), an abelian group structure [h, d, g], where $h = \phi(n)$ is the order (G.no), d (G.cyc) is a k-component row-vector d of integers d_i such that $d_i > 1$, $d_i || d_{i-1}$ for $i \ge 2$ and

and: math: 'g'(: literal: 'G.gen') is a: math: 'k' - component row vector giving generators of the image of the cyclic group of the cyclic group

• *flag* = 1 the result is a bid structure; this allows computing discrete logarithms using znlog (also in the noncyclic case!).

```
? G = znstar(40)
%1 = [16, [4, 2, 2], [Mod(17, 40), Mod(21, 40), Mod(11, 40)]]
? G.no \\ eulerphi(40)
%2 = 16
? G.cyc \\ cycle structure
%3 = [4, 2, 2]
? G.gen \\ generators for the cyclic components
%4 = [Mod(17, 40), Mod(21, 40), Mod(11, 40)]
? apply(znorder, G.gen)
%5 = [4, 2, 2]
```

For user convenience, we define znstar(0) as [2, [2], [-1]], corresponding to \mathbb{Z}^* , but flag = 1 is not implemented in this trivial case.

cypari2.pari_instance.default_bitprec()

Return the default precision in bits.

```
>>> from cypari2.pari_instance import default_bitprec
>>> default_bitprec()
64
```

cypari2.pari_instance.prec_bits_to_dec(prec_in_bits)

Convert from precision expressed in bits to precision expressed in decimal.

Examples:

```
>>> from cypari2.pari_instance import prec_bits_to_dec
>>> prec_bits_to_dec(53)
15
>>> [(32*n, prec_bits_to_dec(32*n)) for n in range(1, 9)]
[(32, 9), (64, 19), (96, 28), (128, 38), (160, 48), (192, 57), (224, 67), (256, 77)]
```

cypari2.pari_instance.prec_bits_to_words(prec_in_bits)

Convert from precision expressed in bits to pari real precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

Examples:

```
>>> from cypari2.pari_instance import prec_bits_to_words
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> prec_bits_to_words(70) == (5 if bitness == '32' else 4)
True
```

cypari2.pari_instance.prec_dec_to_bits(prec_in_dec)

Convert from precision expressed in decimal to precision expressed in bits.

Examples:

cypari2.pari_instance.prec_dec_to_words(prec_in_dec)

Convert from precision expressed in decimal to precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

```
>>> from cypari2.pari_instance import prec_dec_to_words
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> prec_dec_to_words(38) == (6 if bitness == '32' else 4)
True
```

cypari2.pari_instance.prec_words_to_bits(prec_in_words)

Convert from pari real precision expressed in words to precision expressed in bits. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

Examples:

```
>>> from cypari2.pari_instance import prec_words_to_bits
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> prec_words_to_bits(10) == (256 if bitness == '32' else 512)
True
```

```
>>> ans32 = [(3, 32), (4, 64), (5, 96), (6, 128), (7, 160), (8, 192), (9, 224)]
>>> ans64 = [(3, 64), (4, 128), (5, 192), (6, 256), (7, 320), (8, 384), (9, 448)] #______64-bit
>>> [(n, prec_words_to_bits(n)) for n in range(3, 10)] == (ans32 if bitness == '32'______else ans64)
True
```

cypari2.pari_instance.prec_words_to_dec(prec_in_words)

Convert from precision expressed in words to precision expressed in decimal. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

```
>>> from cypari2.pari_instance import prec_words_to_dec
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> prec_words_to_dec(5) == (28 if bitness == '32' else 57)
True
```

TWO

THE GEN CLASS WRAPPING PARI'S GEN TYPE

AUTHORS:

- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-03-06): added newtonpoly
- Justin Walker: contributed some of the function definitions
- Gonzalo Tornaria: improvements to conversions; much better error handling.
- Robert Bradshaw, Jeroen Demeyer, William Stein (2010-08-15): Upgrade to PARI 2.4.3 (Sage ticket #9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (Sage ticket #11611, Sage ticket #11854, Sage ticket #11952)
- Peter Bruin (2013-11-17): move Pari to a separate file (Sage ticket #15185)
- Jeroen Demeyer (2014-02-09): upgrade to PARI 2.7 (Sage ticket #15767)
- Martin von Gagern (2014-12-17): Added some Galois functions (Sage ticket #17519)
- Jeroen Demeyer (2015-01-12): upgrade to PARI 2.8 (Sage ticket #16997)
- Jeroen Demeyer (2015-03-17): automatically generate methods from pari.desc (Sage ticket #17631 and Sage ticket #17860)
- Kiran Kedlaya (2016-03-23): implement infinity type
- Luca De Feo (2016-09-06): Separate Sage-specific components from generic C-interface in Pari (Sage ticket #20241)
- Vincent Delecroix (2017-04-29): Python 3 support and doctest conversion

class cypari2.gen.Gen

Wrapper for a PARI GEN with memory management.

This wraps PARI objects which live either on the PARI stack or on the PARI heap. Results from PARI computations appear on the PARI stack and we try to keep them there. However, when the stack fills up, we copy ("clone" in PARI speak) all live objects from the stack to the heap. This happens transparently for the user.

Ser(*v*, *precision*)

Return a power series or Laurent series in the variable v constructed from the object f.

INPUT:

- **f** PARI gen
- v PARI variable (default: *x*)
- precision the desired relative precision (default: the value returned by pari. get_series_precision()). This is the absolute precision minus the *v*-adic valuation.

OUTPUT:

PARI object of type t_SER

The series is constructed from f in the following way:

- If f is a scalar, a constant power series is returned.
- If *f* is a polynomial, it is converted into a power series in the obvious way.
- If f is a rational function, it will be expanded in a Laurent series around v = 0.
- If f is a vector, its coefficients become the coefficients of the power series, starting from the constant term. This is the convention used by the function Polrev(), and the reverse of that used by Pol().

Warning: This function will not transform objects containing variables of higher priority than v.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> pari(2).Ser()
2 + 0(x^16)
>>> pari('Mod(0, 7)').Ser()
Mod(0, 7)*x^15 + 0(x^16)

```
>>> x = pari([1, 2, 3, 4, 5])
>>> x.Ser()
1 + 2*x + 3*x^2 + 4*x^3 + 5*x^4 + 0(x^16)
>>> f = x.Ser('v'); print(f)
1 + 2*v + 3*v^2 + 4*v^3 + 5*v^4 + 0(v^16)
>>> pari(1)/f
1 - 2*v + v^2 + 6*v^5 - 17*v^6 + 16*v^7 - 5*v^8 + 36*v^10 - 132*v^11 + 181*v^12_
- 110*v^13 + 25*v^14 + 216*v^15 + 0(v^16)
```

```
>>> pari('x^5').Ser(precision=20)
x^5 + 0(x^25)
>>> pari('1/x').Ser(precision=1)
x^-1 + 0(x^0)
```

Str()

Str(self): Return the print representation of self as a PARI object.

INPUT:

```
• self-gen
```

OUTPUT:

• gen - a PARI Gen of type t_STR, i.e., a PARI string

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari([1,2,['abc',1]]).Str()
"[1, 2, [abc, 1]]"
>>> pari([1,1, 1.54]).Str()
"[1, 1, 1.540000000000]"
>>> pari(1).Str()  # 1 is automatically converted to string rep
"1"
>>> x = pari('x')  # PARI variable "x"
>>> x.Str()  # is converted to string rep.
"x"
>>> x.Str().type()
't_STR'
```

Strexpand()

Concatenate the entries of the vector x into a single string, then perform tilde expansion and environment variable expansion similar to shells.

INPUT:

• \mathbf{x} – PARI gen. Either a vector or an element which is then treated like [x].

OUTPUT:

• PARI string (type t_STR)

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari('"~/subdir"').Strexpand()
"..."
>>> pari('"$SHELL"').Strexpand()
"..."
```

Tests:

```
>>> a = pari('"$HOME"')
>>> a.Strexpand() != a
True
```

Strtex()

Strtex(x): Translates the vector x of PARI gens to TeX format and returns the resulting concatenated strings as a PARI t_STR.

INPUT:

• \mathbf{x} – PARI gen. Either a vector or an element which is then treated like [x].

OUTPUT:

• PARI string (type t_STR)

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

Zn_issquare(n)

Return True if self is a square modulo *n*, False if not.

INPUT:

- self integer
- n integer or factorisation matrix

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(3).Zn_issquare(4)
False
>>> pari(4).Zn_issquare(pari(30).factor())
True
```

$Zn_sqrt(n)$

Return a square root of self modulo *n*, if such a square root exists; otherwise, raise a ValueError.

INPUT:

- self integer
- n integer or factorisation matrix

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(3).Zn_sqrt(4)
Traceback (most recent call last):
....
ValueError: 3 is not a square modulo 4
>>> pari(4).Zn_sqrt(pari(30).factor())
22
```

allocatemem(*args)

Do not use this. Use pari.allocatemem() instead.

Tests:

arity()

Return the number of arguments of this t_CLOSURE.

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari("() -> 42").arity()
0
>>> pari("(x) -> x").arity()
1
>>> pari("(x,y,z) -> x+y+z").arity()
3
```

bernfrac()

The Bernoulli number B_x , where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$, *ldots*, expressed as a rational number. The argument *x* should be of type integer.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(18).bernfrac()
43867/798
>>> [pari(n).bernfrac() for n in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
```

bernreal(precision)

The Bernoulli number B_x , as for the function bernfrac, but B_x is returned as a real number (with the current precision).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(18).bernreal()
54.9711779448622
```

besselk(x, precision)

nu.besselk(x): K-Bessel function (modified Bessel function of the second kind) of index nu, which can be complex, and argument x.

If *nu* or *x* is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

INPUT:

- nu a complex number
- x real number (positive or negative)

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> pari(complex(2, 1)).besselk(3)
0.0455907718407551 + 0.0289192946582081*I

```
>>> pari(complex(2, 1)).besselk(-3)
-4.34870874986752 - 5.38744882697109*1
```

```
>>> pari(complex(2, 1)).besselk(300)
3.74224603319728 E-132 + 2.49071062641525 E-134*I
```

bid_get_cyc()

Returns the structure of the group $(O_K/I)^*$, where *I* is the ideal represented by self.

NOTE: self must be a "big ideal" (bid) as returned by idealstar for example.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> i = pari('i')
>>> K = (i**2 + 1).bnfinit()
>>> J = K.idealstar(4*i + 2)
>>> J.bid_get_cyc()
[4, 2]
```

bid_get_gen()

Returns a vector of generators of the group $(O_K/I)^*$, where I is the ideal represented by self.

NOTE: self must be a "big ideal" (bid) with generators, as returned by idealstar with flag = 2.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> i = pari('i')
>>> K = (i**2 + 1).bnfinit()
>>> J = K.idealstar(4*i + 2, 2)
>>> J.bid_get_gen()
[7, [-2, -1]~]

We get an exception if we do not supply flag = 2 to idealstar:

```
>>> J = K.idealstar(3)
>>> J.bid_get_gen()
Traceback (most recent call last):
```

```
PariError: missing bid generators. Use idealstar(,,2)
```

bittest(n)

. . .

bittest(x, long n): Returns bit number n (coefficient of 2^n in binary) of the integer x. Negative numbers behave as if modulo a big power of 2.

INPUT:

• x - Gen (pari integer)

OUTPUT:

• bool - a Python bool

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari(6)
>>> x.bittest(0)
False
>>> x.bittest(1)
True
>>> x.bittest(2)
True
>>> x.bittest(3)
False
>>> pari(-3).bittest(0)
True
>>> pari(-3).bittest(1)
False
>>> [pari(-3).bittest(n) for n in range(10)]
[True, False, True, True, True, True, True, True]
```

bnf_get_cyc()

Returns the structure of the class group of this number field as a vector of SNF invariants.

NOTE: self must be a "big number field" (bnf).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**2 + 65).bnfinit()
>>> K.bnf_get_cyc()
[4, 2]
```

bnf_get_fu()

Return the fundamental units Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
```

```
>>> (x**2 - 65).bnfinit().bnf_get_fu()
[Mod(x - 8, x^2 - 65)]
>>> (x**4 - x**2 + 1).bnfinit().bnf_get_fu()
[Mod(x - 1, x^4 - x^2 + 1)]
>>> p = x**8 - 40*x**6 + 352*x**4 - 960*x**2 + 576
>>> len(p.bnfinit().bnf_get_fu())
7
```

bnf_get_gen()

Returns a vector of generators of the class group of this number field.

NOTE: self must be a "big number field" (bnf).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**2 + 65).bnfinit()
>>> G = K.bnf_get_gen(); G
[[3, 2; 0, 1], [2, 1; 0, 1]]
```

bnf_get_no()

Returns the class number of self, a "big number field" (bnf).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**2 + 65).bnfinit()
>>> K.bnf_get_no()
8
```

bnf_get_reg()

Returns the regulator of this number field.

NOTE: self must be a "big number field" (bnf).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**4 - 4*x**2 + 1).bnfinit()
```

```
>>> K.bnf_get_reg()
2.66089858019037...
```

bnf_get_tu()

Return the torsion unit

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

>>> x = pari('x')

```
>>> for p in [x**2 - 65, x**4 - x**2 + 1, x**8 - 40*x**6 + 352*x**4 - 960*x**2_

→+ 576]:

... bnf = p.bnfinit()

... n, z = bnf.bnf_get_tu()

... if pari.version() < (2,11,0) and z.lift().poldegree() == 0: z = z.lift()

print([p, n, z])

[x*2 - 65, 2, -1]

[x*4 - x*2 + 1, 12, Mod(x, x*4 - x*2 + 1)]

[x*8 - 40*x*6 + 352*x*4 - 960*x*2 + 576, 2, -1]
```

bnfunit()

Deprecated in cypari 2.1.2

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> x = pari('x')

```
>>> import warnings
>>> with warnings.catch_warnings(record=True) as w:
... warnings.simplefilter('always')
... funits = (x**2 - 65).bnfinit().bnfunit()
... assert len(w) == 1
... assert issubclass(w[0].category, DeprecationWarning)
>>> funits
[Mod(x - 8, x^2 - 65)]
```

change_variable_name(var)

In self, which must be a t_POL or t_SER, set the variable to var. If the variable of self is already var, then return self.

Warning: You should be careful with variable priorities when applying this on a polynomial or series of which the coefficients have polynomial components. To be safe, only use this function on polynomials with integer or rational coefficients. For a safer alternative, use subst().

```
>>> pari = Pari()
>>> f = pari('x^3 + 17*x + 3')
>>> f.change_variable_name("y")
y^3 + 17*y + 3
>>> f = pari('1 + 2*y + 0(y^10)')
>>> f.change_variable_name("q")
1 + 2*q + 0(q^10)
>>> f.change_variable_name("y") is f
True
```

>>> from cypari2 import Pari

In PARI, I refers to the square root of -1, so it cannot be used as variable name. Note the difference with subst():

```
>>> f = pari('x^2 + 1')
>>> f.change_variable_name("I")
Traceback (most recent call last):
....
PariError: I already exists with incompatible valence
>>> f.subst("x", "I")
0
```

cmp(right)

Compare self and right.

This uses PARI's cmp_universal() routine, which defines a total ordering on the set of all PARI objects (up to the indistinguishability relation given by gidentical()).

Warning: This comparison is only mathematically meaningful when comparing 2 integers. In particular, when comparing rationals or reals, this does not correspond to the natural ordering.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(5).cmp(pari(5))
0
>>> pari('x^2 + 1').cmp(pari('I-1'))
1
>>> I = pari('I')
>>> I.cmp(I)
0
>>> pari('2/3').cmp(pari('2/5'))
-1
>>> two.cmp(pari(1.0))
1
>>> two.cmp(pari(2.0))
1
>>> two.cmp(pari(3.0))
1
```
```
>>> f = pari("0*ffgen(ffinit(29, 10))")
>>> pari(0).cmp(f)
-1
>>> pari("'x").cmp(f)
1
>>> pari("'x").cmp(0)
Traceback (most recent call last):
....
TypeError: Cannot convert int to cypari2.gen.Gen_base
```

debug(depth)

Show the internal structure of self (like the x command in gp).

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> pari('[1/2, 1 + 1.0*I]').debug()
[&=...] VEC(lg=3):...
1st component = [&=...] FRAC(lg=3):...
num = [&=...] INT(lg=3):... (+,lgefint=3):...
den = [&=...] INT(lg=3):... (+,lgefint=3):...
2nd component = [&=...] COMPLEX(lg=3):...
real = [&=...] INT(lg=3):... (+,lgefint=3):...
imag = [&=...] REAL(lg=...):... (+,expo=0):...
```

disc()

Return the discriminant of this object.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> e = pari([0, -1, 1, -10, -20]).ellinit()
>>> e.disc()
-161051
>>> _.factor()
[-1, 1; 11, 5]
```

eint1(n, precision)

x.eint1(n): exponential integral E1(x):

 $\int_{x}^{\infty} \frac{e^{-t}}{t} dt$

If n is present, output the vector [eint1(x), eint1(2*x), ..., eint1(n*x)]. This is faster than repeatedly calling eint1(i*x).

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

REFERENCE:

• See page 262, Prop 5.6.12, of Cohen's book "A Course in Computational Algebraic Number Theory".

Examples:

ellan(n, python_ints)

Return the first *n* Fourier coefficients of the modular form attached to this elliptic curve. See ellak for more details.

INPUT:

- n a long integer
- python_ints bool (default is False); if True, return a list of Python ints instead of a PARI Gen wrapper.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> e = pari([0, -1, 1, -10, -20]).ellinit()
>>> e.ellan(3)
[1, -2, -1]
>>> e.ellan(20)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]
>>> e.ellan(-1)
[]
>>> v = e.ellan(10, python_ints=True); v
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
>>> type(v)
<... 'list'>
>>> type(v[0])
<... 'int'>
```

ellaplist(n, python_ints)

e.ellaplist(n): Returns a PARI list of all the prime-indexed coefficients a_p (up to n) of the *L*-function of the elliptic curve *e*, i.e. the Fourier coefficients of the newform attached to *e*.

INPUT:

- self an elliptic curve
- n a long integer
- python_ints bool (default is False); if True, return a list of Python ints instead of a PARI Gen wrapper.

Warning: The curve e must be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellaplist (or you will get INCORRECT RESULTS!)

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> e = pari([0, -1, 1, -10, -20]).ellinit()
>>> v = e.ellaplist(10); v
[-2, -1, 1, -2]
>>> type(v)
<... 'cypari2.gen.Gen'>
>>> v.type()
't_VEC'
>>> e.ellan(10)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
>>> v = e.ellaplist(10, python_ints=True); v
[-2, -1, 1, -2]
>>> type(v)
<... 'list'>
>>> type(v[0])
<... 'int'>
```

Tests:

```
>>> v = e.ellaplist(1)
>>> v, type(v)
([], <... 'cypari2.gen.Gen'>)
>>> v = e.ellaplist(1, python_ints=True)
>>> v, type(v)
([], <... 'list'>)
```

ellisoncurve(x)

e.ellisoncurve(x): return True if the point x is on the elliptic curve e, False otherwise.

If the point or the curve have inexact coefficients, an attempt is made to take this into account.

Examples:

>>> pari = Pari()

>>> from cypari2 import Pari

```
>>> e = pari([0,1,1,-2,0]).ellinit()
>>> e.ellisoncurve([1,0])
True
>>> e.ellisoncurve([1,1])
False
>>> e.ellisoncurve([1,0.0000000000000000]))
False
>>> e.ellisoncurve([1,0.0000000000000000])
True
>>> e.ellisoncurve([0])
True
```

ellminimalmodel()

ellminimalmodel(e): return the standard minimal integral model of the rational elliptic curve e and the corresponding change of variables. INPUT:

• e - Gen (that defines an elliptic curve)

OUTPUT:

• gen - minimal model

• gen - change of coordinates

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> e = pari([1,2,3,4,5]).ellinit()
>>> F, ch = e.ellminimalmodel()
>>> F[:5]
[1, -1, 0, 4, 3]
>>> ch
[1, -1, 0, -1]
>>> e.ellchangecurve(ch)[:5]
[1, -1, 0, 4, 3]
```

elltors()

Return information about the torsion subgroup of the given elliptic curve.

INPUT:

• e - elliptic curve over QQ

OUTPUT:

- gen the order of the torsion subgroup, a.k.a. the number of points of finite order
- gen vector giving the structure of the torsion subgroup as a product of cyclic groups, sorted in nonincreasing order
- gen vector giving points on e generating these cyclic groups

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> e = pari([1,0,1,-19,26]).ellinit()
>>> e.elltors()
[12, [6, 2], [[1, 2], [3, -2]]]
```

ellwp(*z*, *n*, *flag*, *precision*)

Return the value or the series expansion of the Weierstrass *P*-function at z on the lattice *self* (or the lattice defined by the elliptic curve *self*).

INPUT:

- self an elliptic curve created using ellinit or a list [om1, om2] representing generators for a lattice.
- z (default: 'z') a complex number or a variable name (as string or PARI variable).
- n (default: 20) if 'z' is a variable, compute the series expansion up to at least $O(z^n)$.
- flag (default = 0): If flag is 0, compute only P(z). If flag is 1, compute [P(z), P'(z)].

OUTPUT:

• *P*(*z*) (if flag is 0) or [*P*(*z*), *P*'(*z*)] (if flag is 1). numbers

Examples:

We first define the elliptic curve $X_0(11)$:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> E = pari([0,-1,1,-10,-20]).ellinit()

Compute P(1):

```
>>> E.ellwp(1)
13.9658695257485
```

Compute P(1+i), where i = sqrt(-1):

```
>>> E.ellwp(pari(complex(1, 1)))
-1.11510682565555 + 2.33419052307470*I
>>> E.ellwp(complex(1, 1))
-1.11510682565555 + 2.33419052307470*I
```

The series expansion, to the default $O(z^20)$ precision:

Compute the series for wp to lower precision:

>>> E.ellwp(n=4) z^-2 + 31/15*z^2 + 0(z^4)

Next we use the version where the input is generators for a lattice:

```
>>> pari([1.2692, complex(0.63, 1.45)]).ellwp(1)
13.9656146936689 + 0.000644829272810...*I
```

With flag=1, compute the pair P(z) and P'(z):

```
>>> E.ellwp(1, flag=1)
[13.9658695257485, 101.123860176015]
```

eval(*args, **kwds)

Evaluate **self** with the given arguments.

This is currently implemented in 3 cases:

- univariate polynomials, rational functions, power series and Laurent series (using a single unnamed argument or keyword arguments),
- any PARI object supporting the PARI function pari:substvec (in particular, multivariate polynomials) using keyword arguments,
- objects of type t_CLOSURE (functions in GP bytecode form) using unnamed arguments.

In no case is mixing unnamed and keyword arguments allowed.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> f = pari('x^2 + 1')
>>> f.type()
't_POL'
>>> f.eval(pari('I'))
0
>>> f.eval(x=2)
5
>>> (1/f).eval(x=1)
1/2
```

The notation f(x) is an alternative for f.eval(x):

>>> f(3) == f.eval(3) True

>>> f = pari('Mod(x^2 + x + 1, 3)')
>>> f(2)
Mod(1, 3)

Evaluating a power series:

>>> f = pari('1 + x + x³ + 0(x⁷)')
>>> f(2*pari('y')**2)
1 + 2*y² + 8*y⁶ + 0(y¹⁴)

Substituting zero is sometimes possible, and trying to do so in illegal cases can raise various errors:

```
>>> pari('1 + 0(x^3)').eval(0)
1
>>> pari('1/x').eval(0)
Traceback (most recent call last):
. . .
PariError: impossible inverse in gdiv: 0
>>> pari('1/x + 0(x^2)').eval(0)
Traceback (most recent call last):
. . .
PariError: impossible inverse in gsubst: 0
>>> pari('1/x + 0(x^2)').eval(pari('0(x^3)'))
Traceback (most recent call last):
. . .
PariError: impossible inverse in ...
>>> pari('0(x^0)').eval(0)
Traceback (most recent call last):
. . .
PariError: forbidden substitution t_SER , t_INT
```

Evaluating multivariate polynomials:

```
>>> f = pari('y^2 + x^3')
>>> f(1)
          # Dangerous, depends on PARI variable ordering
y^{2} + 1
>>> f(x=1) # Safe
y^2 + 1
>>> f(y=1)
x^{3} + 1
>>> f(1, 2)
Traceback (most recent call last):
. . .
TypeError: evaluating PARI t_POL takes exactly 1 argument (2 given)
>>> f(y='x', x='2*y')
x^{2} + 8^{*}y^{3}
>>> f()
x^{3} + y^{2}
```

It's not an error to substitute variables which do not appear:

>>> f.eval(z=37)
x^3 + y^2
>>> pari(42).eval(t=0)
42

We can define and evaluate closures as follows:

```
>>> T = pari('n -> n + 2')
>>> T.type()
't_CLOSURE'
>>> T.eval(3)
5
```

>>> T = pari('() -> 42')
>>> T()
42

```
>>> pr = pari('s -> print(s)')
>>> pr.eval('"hello world"')
hello world
```

```
>>> f = pari('myfunc(x,y) = x*y')
>>> f.eval(5, 6)
30
```

Default arguments work, missing arguments are treated as zero (like in GP):

```
>>> f = pari("(x, y, z=1.0) -> [x, y, z]")
>>> f(1, 2, 3)
[1, 2, 3]
>>> f(1, 2)
[1, 2, 1.000000000000]
>>> f(1)
[1, 0, 1.000000000000]
```

>>> f() [0, 0, 1.0000000000000]

Variadic closures are supported as well (Sage ticket #18623):

>>> f = pari("(v[..])->length(v)")
>>> f('a', f)
2
>>> g = pari("(x,y,z[..])->[x,y,z]")
>>> g(), g(1), g(1,2), g(1,2,3), g(1,2,3,4)
([0, 0, []], [1, 0, []], [1, 2, []], [1, 2, [3]], [1, 2, [3, 4]])

Using keyword arguments, we can substitute in more complicated objects, for example a number field:

```
>>> nf = pari('x^2 + 1').nfinit()
>>> nf
[x^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-38 + 1.0000000000000*I]), [1, 1.
...,00000000000000; 1, -1.000000000000], ..., [2, 0; 0, -2], [2, 0; 0, 2], [1,...,0; 0, -1], [1, [0, -1; 1, 0]], [2]], [0.E-38 + 1.0000000000000*I], [1, x],...,[1, 0; 0, 1], [1, 0, 0, -1; 0, 1, 1, 0]]
>>> nf(x='y')
[y^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-38 + 1.0000000000000*I]), [1, 1.
...,0000000000000; 1, -1.000000000000], ..., [2, 0; 0, -2], [2, 0; 0, 2], [1,...,0; 0, -1], [1, [0, -1; 1, 0]], [2]], [0.E-38 + 1.000000000000*I], [1, y],...,[1, 0; 0, 1], [1, 0, 0, -1; 0, 1, 1, 0]]
```

Tests:

```
>>> T = pari('n -> 1/n')
>>> T.type()
't_CLOSURE'
>>> T(0)
Traceback (most recent call last):
. . .
PariError: _/_: impossible inverse in gdiv: 0
>>> pari('() -> 42')(1,2,3)
Traceback (most recent call last):
. . .
PariError: too many parameters in user-defined function call
>>> pari('n \rightarrow n')(n=2)
Traceback (most recent call last):
. . .
TypeError: cannot evaluate a PARI closure using keyword arguments
>>> pari('x + y')(4, y=1)
Traceback (most recent call last):
. . .
TypeError: mixing unnamed and keyword arguments not allowed when evaluating a_
→PARI object
>>> pari("12345")(4)
Traceback (most recent call last):
. . .
TypeError: cannot evaluate PARI t_INT using unnamed arguments
```

factor(limit, proof)

Return the factorization of x.

INPUT:

- limit (default: -1) is optional and can be set whenever x is of (possibly recursive) rational type. If limit is set, return partial factorization, using primes up to limit.
- proof optional flag. If False (not the default), returned factors larger than 2[/]64} may only be pseudoprimes. If True, always check primality. If not given, use the global PARI default factor_proven which is True by default in cypari.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> pari('x^10-1').factor()
[x - 1, 1; x + 1, 1; x^4 - x^3 + x^2 - x + 1, 1; x^4 + x^3 + x^2 + x + 1, 1]
>>> pari(2**100-1).factor()
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1;
-$8101, 1; 268501, 1]
>>> pari(2**100-1).factor(proof=True)
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1;
-$8101, 1; 268501, 1]
>>> pari(2**100-1).factor(proof=False)
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1;
-$8101, 1; 268501, 1]
```

We illustrate setting a limit:

Setting a limit is invalid when factoring polynomials:

```
>>> pari('x^11 + 1').factor(limit=17)
Traceback (most recent call last):
....
PariError: incorrect type in boundfact (t_POL)
```

factorpadic(p, r)

p-adic factorization of the polynomial pol to precision r.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pol = pari('x^2 - 1')**2
>>> pari(pol).factorpadic(5)
[(1 + 0(5^20))*x + (1 + 0(5^20)), 2; (1 + 0(5^20))*x + (4 + 4*5 + 4*5^2 + 4*5^3]
```

```
\rightarrow 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11 + 4*5^12 +

\rightarrow 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18 + 4*5^19 + 0(5^20)), 2]

>>> pari(pol).factorpadic(5,3)

[(1 + 0(5^3))*x + (1 + 0(5^3)), 2; (1 + 0(5^3))*x + (4 + 4*5 + 4*5^2 + 0(5^3)),]

\rightarrow 2]
```

ffprimroot()

Return a primitive root of the multiplicative group of the definition field of the given finite field element. INPUT:

• self – a PARI finite field element (FFELT)

OUTPUT:

A generator of the multiplicative group of the finite field generated by self.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> b = pari(9).ffgen().ffprimroot()
>>> b.fforder()
8
```

fibonacci()

Return the Fibonacci number of index x.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(18).fibonacci()
2584
>>> [pari(n).fibonacci() for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

galoissubfields(flag, v)

List all subfields of the Galois group self.

This wraps the galoissubfields function from PARI.

This method is essentially the same as applying galoisfixedfield() to each group returned by galoissubgroups().

INPUT:

- self A Galois group as generated by galoisinit().
- flag Has the same meaning as in galoisfixedfield().
- v Has the same meaning as in galoisfixedfield().

OUTPUT:

A vector of all subfields of this group. Each entry is as described in the galoisfixedfield() method. Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> G = pari('x^6 + 108').galoisinit()
>>> G.galoissubfields(flag=1)
[x, x^2 + 972, x^3 + 54, x^3 + 864, x^3 - 54, x^6 + 108]
>>> G = pari('x^4 + 1').galoisinit()
>>> G.galoissubfields(flag=2, v='z')[3]
[...^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - z*x - 1, x^2 + z*x - 1]]
```

gequal(b)

Check whether *a* and *b* are equal using PARI's gequal.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> a = pari(1); b = pari(1.0); c = pari('"some_string"')
>>> a.gequal(a)
True
>>> b.gequal(b)
True
>>> c.gequal(c)
True
>>> a.gequal(b)
True
>>> a.gequal(c)
False
```

WARNING: this relation is not transitive:

```
>>> a = pari('[0]'); b = pari(0); c = pari('[0,0]')
>>> a.gequal(b)
True
>>> b.gequal(c)
True
>>> a.gequal(c)
False
```

gequal0()

Check whether a is equal to zero.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(0).gequal0()
True
>>> pari(1).gequal0()
False
>>> pari(1e-100).gequal0()
```

```
False
>>> pari("0.0 + 0.0*I").gequal0()
True
>>> (pari('ffgen(3^20)')*0).gequal0()
True
```

gequal_long(b)

Check whether *a* is equal to the long int *b* using PARI's gequalsg.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> a = pari(1); b = pari(2.0); c = pari('3*matid(3)')
>>> a.gequal_long(1)
True
>>> a.gequal_long(-1)
False
>>> b.gequal_long(0)
False
>>> b.gequal_long(2)
True
>>> b.gequal_long(-2)
False
>>> c.gequal_long(3)
True
>>> c.gequal_long(-3)
False
```

getattr(attr)

Return the PARI attribute with the given name.

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> K = pari("nfinit(x<sup>2</sup> - x - 1)")
>>> K.getattr("pol")
x<sup>2</sup> - x - 1
>>> K.getattr("disc")
5
```

```
>>> K.getattr("reg")
Traceback (most recent call last):
...
PariError: _.reg: incorrect type in reg (t_VEC)
>>> K.getattr("zzz")
Traceback (most recent call last):
...
PariError: not a function in function call
```

idealmoddivisor(ideal)

Return a 'small' ideal equivalent to ideal in the ray class group that the bnr structure self encodes.

INPUT:

- self a bnr structure as outputted from bnrinit.
- ideal an ideal in the underlying number field of the bnr structure.

OUTPUT: An ideal representing the same ray class as ideal but with 'small' generators. If ideal is not coprime to the modulus of the bnr, this results in an error.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> i = pari('i')
>>> K = (i**4 - 2).bnfinit()
>>> R = K.bnrinit(5,1)
>>> R.idealmoddivisor(K[6][6][1])
[2, 0, 0, 0; 0, 1, 0, 0; 0, 0, 1, 0; 0, 0, 0, 1]
>>> R.idealmoddivisor(K.idealhnf(5))
Traceback (most recent call last):
...
PariError: elements not coprime in idealaddtoone:
    [5, 0, 0, 0; 0, 5, 0, 0; 0, 0, 5, 0; 0, 0, 0, 5]
    [5, 0, 0, 0; 0, 5, 0, 0; 0, 0, 5, 0; 0, 0, 0, 5]
```

ispower(k)

Determine whether or not self is a perfect k-th power. If k is not specified, find the largest k so that self is a k-th power.

INPUT:

• k - int (optional)

OUTPUT:

- power int, what power it is
- g what it is a power of

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(9).ispower()
(2, 3)
>>> pari(17).ispower()
(1, 17)
>>> pari(17).ispower(2)
(False, None)
>>> pari(17).ispower(1)
(1, 17)
>>> pari(2).ispower()
(1, 2)
```

isprime(flag)

Return True if x is a PROVEN prime number, and False otherwise.

INPUT:

• flag – If flag is 0 or omitted, use a combination of algorithms. If flag is 1, the primality is certified by the Pocklington-Lehmer Test. If flag is 2, the primality is certified using the APRCL test. If flag is 3, use ECPP.

OUTPUT: bool

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(9).isprime()
False
>>> pari(17).isprime()
True
                      # smallest Carmichael number
>>> n = pari(561)
>>> n.isprime()
                      # not just a pseudo-primality test!
False
>>> n.isprime(1)
False
>>> n.isprime(2)
False
>>> n = pari(2^{**}31 - 1)
>>> n.isprime(1)
True
```

isprimepower()

Check whether self is a prime power (with an exponent ≥ 1).

INPUT:

self - A PARI integer

OUTPUT:

A tuple (k, p) where k is a Python integer and p a PARI integer.

- If the input was a prime power, *p* is the prime and *k* the power.
- Otherwise, *k* = 0 and *p* is self.

See also:

If you don't need a proof that p is prime, you can use *ispseudoprimepower()* instead.

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(9).isprimepower()
(2, 3)
>>> pari(17).isprimepower()
(1, 17)
>>> pari(18).isprimepower()
(0, 18)
>>> pari(3**12345).isprimepower()
(12345, 3)
```

ispseudoprime(flag)

ispseudoprime(x, flag=0): Returns True if x is a pseudo-prime number, and False otherwise.

INPUT:

flag - int 0 (default): checks whether x is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence (P,-1), P smallest positive integer such that P^2 - 4 is not a square mod x). 0: checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

OUTPUT:

• bool - True or False, or when flag=1, either False or a tuple (True, cert) where cert is a primality certificate.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(9).ispseudoprime()
False
>>> pari(17).ispseudoprime()
True
>>> n = pari(561)  # smallest Carmichael number
>>> n.ispseudoprime(2)
False
```

ispseudoprimepower()

Check whether self is the power (with an exponent >= 1) of a pseudo-prime.

INPUT:

• self - A PARI integer

OUTPUT:

A tuple (k, p) where k is a Python integer and p a PARI integer.

- If the input was a pseudoprime power, p is the pseudoprime and k the power.
- Otherwise, k = 0 and p is self.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(3**12345).ispseudoprimepower()
(12345, 3)
>>> p = pari(2**1500 + 1465)  # nextprime(2^1500)
>>> (p**11).ispseudoprimepower()[0] # very fast
11
```

issquare(find_root)

issquare(x,n): True if x is a square, False if not. If find_root is given, also returns the exact square root.

issquarefree()

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(10).issquarefree()
True
>>> pari(20).issquarefree()
False
```

j()

Return the j-invariant of this object.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> e = pari([0, -1, 1, -10, -20]).ellinit()
>>> e.j()
-122023936/161051
>>> _.factor()
[-1, 1; 2, 12; 11, -5; 31, 3]
```

lift_centered(v)

Same as lift, except that t_INTMOD and t_PADIC components are lifted using centered residues:

- for a t_INTMOD $x \in \mathbb{Z}/n\mathbb{Z}$, the lift y is such that -n/2 < y <= n/2.
- a t_PADIC x is lifted in the same way as above (modulo $p^padicprec(x)$) if its valuation v is nonnegative; if not, returns the fraction p^v centerlift (xp^{-v}) ; in particular, rational reconstruction is not attempted. Use bestappr for this.

For backward compatibility, centerlift(x, 'v) is allowed as an alias for lift(x, 'v).

list()

Convert self to a Python list with Gen components.

Examples:

A PARI vector becomes a Python list:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> L = pari("vector(10,i,i^2)").list()
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> type(L)
<... 'list'>
>>> type(L[0])
<... 'cypari2.gen.Gen'>
```

For polynomials, list() returns the list of coefficients:

```
>>> pol = pari("x^3 + 5/3*x"); pol.list()
[0, 5/3, 0, 1]
```

For power series or Laurent series, we get all coefficients starting from the lowest degree term. This includes trailing zeros:

>>> pari('x^2 + 0(x^8)').list()
[1, 0, 0, 0, 0, 0]
>>> pari('x^-2 + 0(x^0)').list()
[1, 0]

For matrices, we get a list of columns:

```
>>> M = pari.matrix(3,2,[1,4,2,5,3,6]); M
[1, 4; 2, 5; 3, 6]
>>> M.list()
[[1, 2, 3]~, [4, 5, 6]~]
```

log_gamma(precision)

Principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with nonpositive integers removed, and can have much larger arguments than gamma itself.

For x a power series such that x(0) is not a pole of gamma, compute the Taylor expansion. (PARI only knows about regular power series and can't include logarithmic terms.)

```
? lngamma(1+x+0(x^2))
%1 = -0.57721566490153286060651209008240243104*x + 0(x^2)
? lngamma(x+0(x^2))
*** at top-level: lngamma(x+0(x^2))
*** ^------
*** lngamma: domain error in lngamma: valuation != 0
? lngamma(-1+x+0(x^2))
*** lngamma: Warning: normalizing a series with 0 leading term.
*** at top-level: lngamma(-1+x+0(x^2))
*** ^------
*** lngamma: domain error in intformal: residue(series, pole) != 0
```

matkerint(flag)

Return the integer kernel of a matrix.

This is the LLL-reduced Z-basis of the kernel of the matrix x with integral entries.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari('[2,1;2,1]').matkerint()
[1; -2]
>>> import warnings
>>> with warnings.catch_warnings(record=True) as w:
... warnings.simplefilter('always')
... pari('[2,1;2,1]').matkerint(1)
... assert len(w) == 1
```

```
assert issubclass(w[0].category, DeprecationWarning)
[1; -2]
```

mattranspose()

Transpose of the matrix self.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> pari('[1,2,3; 4,5,6; 7,8,9]').mattranspose()
[1, 4, 7; 2, 5, 8; 3, 6, 9]
```

Unlike PARI, this always returns a matrix:

```
>>> pari('[1,2,3]').mattranspose()
[1; 2; 3]
>>> pari('[1,2,3]~').mattranspose()
Mat([1, 2, 3])
```

mod()

Given an INTMOD or POLMOD Mod(a,m), return the modulus m.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(4).Mod(5).mod()
5
>>> pari("Mod(x, x*y)").mod()
y*x
>>> pari("[Mod(4,5)]").mod()
Traceback (most recent call last):
...
TypeError: Not an INTMOD or POLMOD in mod()
```

multiplicative_order(o)

x must be an integer mod n, and the result is the order of x in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. Returns an error if x is not invertible. The parameter o, if present, represents a nonzero multiple of the order of x, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord = eulerphi(n) is the cardinality of the group.

ncols()

Return the number of columns of self.

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> pari('matrix(19,8)').ncols()
8

nextprime(add_one)

nextprime(x): smallest pseudoprime greater than or equal to x. If add_one is non-zero, return the smallest pseudoprime strictly greater than x.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> pari(1).nextprime()
2
>>> pari(2).nextprime()
2
>>> pari(2).nextprime(add_one = 1)
3
>>> pari(2**100).nextprime()
1267650600228229401496703205653
```

nf_get_diff()

Returns the different of this number field as a PARI ideal.

INPUT:

• self - A PARI number field being the output of nfinit(), bnfinit() or bnrinit().

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

>>> x = pari('x')
>>> K = (x**4 - 4*x**2 + 1).nfinit()
>>> K.nf_get_diff()
[12, 0, 0, 0; 0, 12, 8, 0; 0, 0, 4, 0; 0, 0, 0, 4]

nf_get_pol()

Returns the defining polynomial of this number field.

INPUT:

```
• self - A PARI number field being the output of nfinit(),
bnfinit() or bnrinit().
```

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**4 - 4*x**2 + 1).bnfinit()
>>> bnr = K.bnrinit(2*x)
```

```
>>> bnr.nf_get_pol()
x^4 - 4*x^2 + 1
```

For relative number fields, this returns the relative polynomial:

>>> y = pari.varhigher('y')
>>> L = K.rnfinit(y**2 - 5)
>>> L.nf_get_pol()
y^2 - 5

An error is raised for invalid input:

```
>>> pari("[0]").nf_get_pol()
Traceback (most recent call last):
....
PariError: incorrect type in pol (t_VEC)
```

nf_get_sign()

Returns a Python list [r1, r2], where r1 and r2 are Python ints representing the number of real embeddings and pairs of complex embeddings of this number field, respectively.

INPUT:

```
    self - A PARI number field being the output of nfinit(),
bnfinit() or bnrinit().
```

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**4 - 4*x**2 + 1).nfinit()
>>> s = K.nf_get_sign(); s
[4, 0]
>>> type(s); type(s[0])
<... 'list'>
<... 'list'>
>>> pari.polcyclo(15).nfinit().nf_get_sign()
[0, 4]
```

nf_get_zk()

Returns a vector with a ZZ-basis for the ring of integers of this number field. The first element is always 1.

INPUT:

```
• self - A PARI number field being the output of nfinit(),
bnfinit() or bnrinit().
```

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> x = pari('x')
>>> K = (x**4 - 4*x**2 + 1).nfinit()
>>> K.nf_get_zk()
[1, x, x^3 - 4*x, x^2 - 2]
```

nf_subst(z)

Given a PARI number field self, return the same PARI number field but in the variable z.

INPUT:

 self - A PARI number field being the output of nfinit(), bnfinit() or bnrinit().

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

>>> K = pari('y^2 + 5').nfinit()

We can substitute in a PARI nf structure:

```
>>> K.nf_get_pol()
y^2 + 5
>>> L = K.nf_subst('a')
>>> L.nf_get_pol()
a^2 + 5
```

We can also substitute in a PARI bnf structure:

```
>>> K = pari('y^2 + 5').bnfinit()
>>> K.nf_get_pol()
y^2 + 5
>>> K.bnf_get_cyc() # Structure of class group
[2]
>>> L = K.nf_subst('a')
>>> L.nf_get_pol()
a^2 + 5
>>> L.bnf_get_cyc() # We still have a bnf after substituting
[2]
```

nfbasis(flag, fa)

Integral basis of the field QQ[a], where a is a root of the polynomial x.

INPUT:

- flag: if set to 1 and fa is not given: assume that no square of a prime > 500000 divides the discriminant of x.
- fa: If present, encodes a subset of primes at which to check for maximality. This must be one of the three following things:
 - an integer: check all primes up to fa using trial division.
 - a vector: a list of primes to check.
 - a matrix: a partial factorization of the discriminant of x.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari('x^3 - 17').nfbasis()
```

 $[1, x, 1/3*x^2 - 1/3*x + 1/3]$

We test flag = 1, noting it gives a wrong result when the discriminant (-4 * $p ^{2}$ * q in the example below) has a big square factor:

```
>>> p = pari(10**10).nextprime(); q = (p+1).nextprime()
>>> x = pari('x'); f = x**2 + p**2*q
>>> pari(f).nfbasis(1)  # Wrong result
[1, x]
>>> pari(f).nfbasis()  # Correct result
[1, 1/10000000019*x]
>>> pari(f).nfbasis(fa=10**6)  # Check primes up to 10^6: wrong result
[1, x]
>>> pari(f).nfbasis(fa="[2,2; %s,2]"%p)  # Correct result and faster
[1, 1/10000000019*x]
>>> pari(f).nfbasis(fa=[2,p])  # Equivalent with the above
[1, 1/10000000019*x]
```

The following alternative syntax closer to PARI/GP can be used

```
>>> pari.nfbasis([f, 1])
[1, x]
>>> pari.nfbasis(f)
[1, 1/1000000019*x]
>>> pari.nfbasis([f, 10**6])
[1, x]
>>> pari.nfbasis([f, "[2,2; %s,2]"%p])
[1, 1/1000000019*x]
>>> pari.nfbasis([f, [2,p]])
[1, 1/1000000019*x]
```

nfbasis_d(flag, fa)

Like *nfbasis()*, but return a tuple (B, D) where B is the integral basis and D the discriminant.

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> F = pari('x^3 - 2').nfinit()
>>> F[0].nfbasis_d()
([1, x, x^2], -108)
```

```
>>> G = pari('x^5 - 11').nfinit()
>>> G[0].nfbasis_d()
([1, x, x^2, x^3, x^4], 45753125)
```

```
>>> pari([-2,0,0,1]).Polrev().nfbasis_d()
([1, x, x^2], -108)
```

nfbasistoalg_lift(x)

Transforms the column vector \mathbf{x} on the integral basis into a polynomial representing the algebraic number.

INPUT:

- nf a number field
- \mathbf{x} a column of rational numbers of length equal to the degree of $\mathbf{n}\mathbf{f}$ or a single rational number

OUTPUT:

• nf.nfbasistoalg(x).lift()

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> K = pari('x^3 - 17').nfinit()
>>> K.nf_get_zk()
[1, 1/3*x^2 - 1/3*x + 1/3, x]
>>> K.nfbasistoalg_lift(42)
42
>>> K.nfbasistoalg_lift("[3/2, -5, 0]~")
-5/3*x^2 + 5/3*x - 1/6
>>> K.nf_get_zk() * pari("[3/2, -5, 0]~")
-5/3*x^2 + 5/3*x - 1/6
```

nfeltval(*x*, *p*)

Return the valuation of the number field element x at the prime p.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> nf = pari('x^2 + 1').nfinit()
>>> p = nf.idealprimedec(5)[0]
>>> nf.nfeltval('50 - 25*x', p)
```

nrows()

3

Return the number of rows of self.

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari('matrix(19,8)').nrows()
19
```

omega()

Return the basis for the period lattice of this elliptic curve.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> e = pari([0, -1, 1, -10, -20]).ellinit()
>>> e.omega()
[1.26920930427955, 0.634604652139777 - 1.45881661693850*I]
```

The precision is determined by the ellinit call:

```
>>> e = pari([0, -1, 1, -10, -20]).ellinit(precision=256)
>>> e.omega().bitprecision()
256
```

This also works over quadratic imaginary number fields:

```
>>> e = pari.ellinit([0, -1, 1, -10, -20], "nfinit(y^2 - 2)")
>>> if pari.version() >= (2, 10, 1):
... w = e.omega()
```

padicprime()

The uniformizer of the p-adic ring this element lies in, as a t_INT.

INPUT:

• x - gen, of type t_PADIC

OUTPUT:

• p - gen, of type t_INT

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> y = pari('11^-10 + 5*11^-7 + 11^-6 + 0(11)')
>>> y.padicprime()
11
>>> y.padicprime().type()
't_INT'
```

polinterpolate(ya, x)

self.polinterpolate(ya,x,e): polynomial interpolation at x according to data vectors self, ya (i.e. return P such that P(self[i]) = ya[i] for all i). Also return an error estimate on the returned value.

polred(*args, **kwds)

This function is *deprecated*, use *polredbest()* instead.

polylog(m, flag, precision)

x.polylog(m,flag=0): m-th polylogarithm of x. flag is optional, and can be 0: default, 1: D_m -modified m-th polylog of x, 2: D_m-modified m-th polylog of x, 3: P_m-modified m-th polylog of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

TODO: Add more explanation, copied from the PARI manual.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(10).polylog(3)
5.64181141475134 - 8.32820207698027*I
>>> pari(10).polylog(3,0)
5.64181141475134 - 8.32820207698027*I
>>> pari(10).polylog(3,1)
0.523778453502411
>>> pari(10).polylog(3,2)
-0.400459056163451
```

pr_get_e()

Returns the ramification index (over QQ) of this prime ideal.

NOTE: self must be a PARI prime ideal (as returned by idealprimedec for example).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> i = pari('i')
>>> K = (i**2 + 1).nfinit()
>>> K.idealprimedec(2)[0].pr_get_e()
2
>>> K.idealprimedec(3)[0].pr_get_e()
1
>>> K.idealprimedec(5)[0].pr_get_e()
1
```

pr_get_f()

Returns the residue class degree (over QQ) of this prime ideal.

NOTE: self must be a PARI prime ideal (as returned by idealprimedec for example).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> i = pari('i')
>>> K = (i**2 + 1).nfinit()
>>> K.idealprimedec(2)[0].pr_get_f()
1
>>> K.idealprimedec(3)[0].pr_get_f()
2
```

```
>>> K.idealprimedec(5)[0].pr_get_f()
1
```

pr_get_gen()

Returns the second generator of this PARI prime ideal, where the first generator is self.pr_get_p().

NOTE: self must be a PARI prime ideal (as returned by idealprimedec for example).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> i = pari('i')
>>> K = (i**2 + 1).nfinit()
>>> g = K.idealprimedec(2)[0].pr_get_gen(); g
[1, 1]~
>>> g = K.idealprimedec(3)[0].pr_get_gen(); g
[3, 0]~
>>> g = K.idealprimedec(5)[0].pr_get_gen(); g
[-2, 1]~
```

pr_get_p()

Returns the prime of ZZ lying below this prime ideal.

NOTE: self must be a PARI prime ideal (as returned by idealprimedec for example).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> i = pari('i')
>>> K = (i**2 + 1).nfinit()
>>> F = K.idealprimedec(5); F
[[5, [-2, 1]~, 1, 1, [2, -1; 1, 2]], [5, [2, 1]~, 1, 1, [-2, -1; 1, -2]]]
>>> F[0].pr_get_p()
5
```

python()

Return the closest Python equivalent of the given PARI object.

See gen_to_python() for more informations.

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari('1.2').python()
1.2
>>> pari('389/17').python()
Fraction(389, 17)
```

python_list()

Return a Python list of the PARI gens. This object must be of type t_VEC or t_COL.

INPUT: None

OUTPUT:

• list - Python list whose elements are the elements of the input gen.

Examples:

>>> from cypari2 import Pari
>>> pari = Pari()

```
>>> v = pari([1,2,3,10,102,10])
>>> w = v.python_list()
>>> w
[1, 2, 3, 10, 102, 10]
>>> type(w[0])
<... 'cypari2.gen.Gen'>
>>> pari("[1,2,3]").python_list()
[1, 2, 3]
```

```
>>> pari("[1,2,3]~").python_list()
[1, 2, 3]
```

python_list_small()

Return a Python list of the PARI gens. This object must be of type t_VECSMALL, and the resulting list contains python 'int's.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> v=pari([1,2,3,10,102,10]).Vecsmall()
>>> w = v.python_list_small()
>>> w
[1, 2, 3, 10, 102, 10]
>>> type(w[0])
<... 'int'>
```

qfrep(B, flag)

Vector of (half) the number of vectors of norms from 1 to *B* for the integral and definite quadratic form self. Binary digits of flag mean 1: count vectors of even norm from 1 to 2*B*, 2: return a t_VECSMALL instead of a t_VEC (which is faster).

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> M = pari("[5,1,1;1,3,1;1,1,1]")
>>> M.qfrep(20)
[1, 1, 2, 2, 2, 4, 4, 3, 3, 4, 2, 4, 6, 0, 4, 6, 4, 5, 6, 4]
```

```
>>> M.qfrep(20, flag=1)
[1, 2, 4, 3, 4, 4, 0, 6, 5, 4, 12, 4, 4, 8, 0, 3, 8, 6, 12, 12]
>>> M.qfrep(20, flag=2)
Vecsmall([1, 1, 2, 2, 2, 4, 4, 3, 3, 4, 2, 4, 6, 0, 4, 6, 4, 5, 6, 4])
```

round(estimate)

round(x,estimate=False): If x is a real number, returns x rounded to the nearest integer (rounding up). If the optional argument estimate is True, also returns the binary exponent e of the difference between the original and the rounded value (the "fractional part") (this is the integer ceiling of $\log_2(error)$).

When x is a general PARI object, this function returns the result of rounding every coefficient at every level of PARI object. Note that this is different than what the truncate function does (see the example below).

One use of round is to get exact results after a long approximate computation, when theory tells you that the coefficients must be integers.

INPUT:

- x gen
- estimate (optional) bool, False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded version of x and error estimate in bits, both as gens.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari('1.5').round()
2
>>> pari('1.5').round(True)
(2, -1)
>>> pari('1.5 + 2.1*I').round()
2 + 2*I
>>> pari('1.0001').round(True)
(1, -14)
>>> pari('(2.4*x^2 - 1.7)/x').round()
(2*x^2 - 2)/x
>>> pari('(2.4*x^2 - 1.7)/x').truncate()
2.400000000000*x
```

sage(locals)

Return the closest SageMath equivalent of the given PARI object.

INPUT:

locals – optional dictionary used in fallback cases that involve sage_eval

See gen_to_sage() for more information.

sizebyte()

Return the total number of bytes occupied by the complete tree of the object x. Note that this number depends on whether the computer is 32-bit or 64-bit.

INPUT:

• x - gen

OUTPUT: int (a Python int)

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> pari('1').sizebyte() == (12 if bitness == '32' else 24)
True
```

sizeword()

Return the total number of machine words occupied by the complete tree of the object x. A machine word is 32 or 64 bits, depending on the computer.

INPUT:

• x - gen

OUTPUT: int (a Python int)

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari('0').sizeword()
2
>>> pari('1').sizeword()
3
>>> pari('1000000').sizeword()
3
```

```
>>> import sys
>>> bitness = '64' if sys.maxsize > (1 << 32) else '32'
>>> pari('10^100').sizeword() == (13 if bitness == '32' else 8)
True
>>> pari(1.0).sizeword() == (4 if bitness == '32' else 3)
True
```

```
>>> pari('x + 1').sizeword()
10
>>> pari('[x + 1, 1]').sizeword()
16
```

sqrtn(n, precision)

x.sqrtn(n): return the principal branch of the n-th root of x, i.e., the one such that arg(sqrt(x)) in]-pi/n, pi/n]. Also returns a second argument which is a suitable root of unity allowing one to recover all the other roots. If it was not possible to find such a number, then this second return value is 0. If the argument is present and no square root exists, return 0 instead of raising an error.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

Note: intmods (modulo a prime) and *p*-adic numbers are allowed as arguments.

INPUT:

- x gen
- n integer

OUTPUT:

- gen principal n-th root of x
- gen root of unity z that gives the other roots

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> s, z = pari(2).sqrtn(5)
>>> z
0.309016994374947 + 0.951056516295154*I
>>> s
1.14869835499704
>>> s**5
2.0000000000000
>>> (s*z)**5
2.000000000000 + 0.E-19*I
```

sumdiv()

Return the sum of the divisors of *n*.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(10).sumdiv()
18
```

sumdivk(k)

Return the sum of the k-th powers of the divisors of n.

```
>>> from cypari2 import Pari
>>> pari = Pari()
>>> pari(10).sumdivk(2)
130
```

truncate(estimate)

truncate(x,estimate=False): Return the truncation of x. If estimate is True, also return the number of error bits.

When x is in the real numbers, this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and truncated value (the "fractional part"). If x is a rational function, the result is the integer part (Euclidean quotient of numerator by denominator) and if requested the error estimate is 0.

When truncate is applied to a power series (in X), it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

INPUT:

- x gen
- estimate (optional) bool, which is False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded version of x and error estimate in bits, both as gens.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari('(x^2+1)/x').round()
(x^2 + 1)/x
>>> pari('(x^2+1)/x').truncate()
х
>>> pari('1.043').truncate()
1
>>> pari('1.043').truncate(True)
(1, -5)
>>> pari('1.6').truncate()
1
>>> pari('1.6').round()
2
>>> pari('1/3 + 2 + 3^2 + 0(3^3)').truncate()
34/3
>>> pari('sin(x+0(x^10))').truncate()
1/362880*x^9 - 1/5040*x^7 + 1/120*x^5 - 1/6*x^3 + x
>>> pari('sin(x+0(x^{10}))').round() # each coefficient has abs < 1
x + 0(x^{10})
```

type()

Return the PARI type of self as a string.

Note: In Cython, it is much faster to simply use typ(self.g) for checking PARI types.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari(7).type()
't_INT'
>>> pari('x').type()
't_POL'
>>> pari('oo').type()
't_INFINITY'
```

vecmax()

Return the maximum of the elements of the vector/matrix *x*.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari([1, '-5/3', 8.0]).vecmax()
8.000000000000000
```

vecmin()

Return the minimum of the elements of the vector/matrix *x*.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

```
>>> pari([1, '-5/3', 8.0]).vecmin()
-5/3
```

class cypari2.gen.Gen_base

Wrapper for a PARI GEN containing auto-generated methods.

This class does not manage the GEN inside in any way. It is just a dumb wrapper. In particular, it might be invalid if the GEN is on the PARI stack and the PARI stack has been garbage collected.

You almost certainly want to use one of the derived class *Gen* instead. That being said, *Gen_base* can be used by itself to pass around a temporary GEN within Python where we cannot use C calls.

Col(n)

Transforms the object x into a column vector. The dimension of the resulting vector can be optionally specified via the extra parameter n.

If n is omitted or 0, the dimension depends on the type of x; the vector has a single component, except when x is

• a vector or a quadratic form (in which case the resulting vector is simply the initial object considered as a row vector),

- a polynomial or a power series. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree. In this last case, Vec is the reciprocal function of Pol and Ser respectively,
- a matrix (the column of row vector comprising the matrix is returned),
- a character string (a vector of individual characters is returned).

In the last two cases (matrix and character string), n is meaningless and must be omitted or an error is raised. Otherwise, if n is given, 0 entries are appended at the end of the vector if n > 0, and prepended at the beginning if n < 0. The dimension of the resulting vector is ||n||.

See ??Vec for examples.

Colrev(n)

As Col(x, -n), then reverse the result. In particular, Colrev is the reciprocal function of Polrev: the coefficients of the vector start with the constant coefficient of the polynomial and the others follow by increasing degree.

List()

Transforms a (row or column) vector x into a list, whose components are the entries of x. Similarly for a list, but rather useless in this case. For other types, creates a list with the single element x.

Map()

A "Map" is an associative array, or dictionary: a data type composed of a collection of (key, value) pairs, such that each key appears just once in the collection. This function converts the matrix $[a_1, b_1; a_2, b_2; ...; a_n, b_n]$ to the map $a_i : -- > b_i$.

? M = Map(factor(13!)); ? mapget(M,3) %2 = 5

If the argument x is omitted, creates an empty map, which may be filled later via mapput.

Mat()

Transforms the object x into a matrix. If x is already a matrix, a copy of x is created. If x is a row (resp. column) vector, this creates a 1-row (resp. 1-column) matrix, *unless* all elements are column (resp. row) vectors of the same length, in which case the vectors are concatenated sideways and the attached big matrix is returned. If x is a binary quadratic form, creates the attached $2x^2$ matrix. Otherwise, this creates a $1x^1$ matrix containing x.

```
? Mat(x + 1)
%1 =
[x + 1]
? Vec( matid(3) )
%2 = [[1, 0, 0]~, [0, 1, 0]~, [0, 0, 1]~]
? Mat(%)
%3 =
[1 0 0]
[0 1 0]
[0 0 1]
? Col( [1,2; 3,4] )
%4 = [[1, 2], [3, 4]]~
```

%5 = [1 2] [3 4] ? Mat(Qfb(1,2,3)) %6 = [1 1] [1 3]

Mod(b)

In its basic form, create an intmod or a polmod (*amodb*); *b* must be an integer or a polynomial. We then obtain a t_INTMOD and a t_POLMOD respectively:

```
? t = Mod(2,17); t^8
%1 = Mod(1, 17)
? t = Mod(x,x^2+1); t^2
%2 = Mod(-1, x^2+1)
```

If a%b makes sense and yields a result of the appropriate type (t_INT or scalar/t_POL), the operation succeeds as well:

```
? Mod(1/2, 5)
%3 = Mod(3, 5)
? Mod(7 + 0(3^6), 3)
%4 = Mod(1, 3)
? Mod(Mod(1,12), 9)
%5 = Mod(1, 3)
? Mod(1/x, x^2+1)
%6 = Mod(-x, x^2+1)
? Mod(exp(x), x^4)
%7 = Mod(1/6*x^3 + 1/2*x^2 + x + 1, x^4)
```

If a is a complex object, "base change" it to $\mathbb{Z}/b\mathbb{Z}$ or K[x]/(b), which is equivalent to, but faster than, multiplying it by Mod(1,b):

```
? Mod([1,2;3,4], 2)
%8 =
[Mod(1, 2) Mod(0, 2)]
[Mod(1, 2) Mod(0, 2)]
? Mod(3*x+5, 2)
%9 = Mod(1, 2)*x + Mod(1, 2)
? Mod(x^2 + y*x + y^3, y^2+1)
%10 = Mod(1, y^2 + 1)*x^2 + Mod(y, y^2 + 1)*x + Mod(-y, y^2 + 1)
```

This function is not the same as $x \ % y$, the result of which has no knowledge of the intended modulus y. Compare

```
? x = 4 % 5; x + 1
%11 = 5
? x = Mod(4,5); x + 1
%12 = Mod(0,5)
```

Note that such "modular" objects can be lifted via lift or centerlift. The modulus of a t_INTMOD or t_POLMOD z can be recovered via :math: `z.mod`.

Pol(v)

Transforms the object t into a polynomial with main variable v. If t is a scalar, this gives a constant polynomial. If t is a power series with nonnegative valuation or a rational function, the effect is similar to truncate, i.e. we chop off the $O(X^k)$ or compute the Euclidean quotient of the numerator by the denominator, then change the main variable of the result to v.

The main use of this function is when t is a vector: it creates the polynomial whose coefficients are given by t, with t[1] being the leading coefficient (which can be zero). It is much faster to evaluate Pol on a vector of coefficients in this way, than the corresponding formal expression $a_n X^n + ... + a_0$, which is evaluated naively exactly as written (linear versus quadratic time in n). Polrev can be used if one wants x[1] to be the constant coefficient:

```
? Pol([1,2,3])
%1 = x^2 + 2*x + 3
? Polrev([1,2,3])
%2 = 3*x^2 + 2*x + 1
```

The reciprocal function of Pol (resp. Polrev) is Vec (resp. Vecrev).

? Vec(Pol([1,2,3]))
%1 = [1, 2, 3]
? Vecrev(Polrev([1,2,3]))
%2 = [1, 2, 3]

Warning. This is *not* a substitution function. It will not transform an object containing variables of higher priority than *v*.

```
? Pol(x + y, y)
 *** at top-level: Pol(x+y,y)
 *** ^------
 *** Pol: variable must have higher priority in gtopoly.
```

Polrev(v)

Transform the object t into a polynomial with main variable v. If t is a scalar, this gives a constant polynomial. If t is a power series, the effect is identical to truncate, i.e. it chops off the $O(X^k)$.

The main use of this function is when t is a vector: it creates the polynomial whose coefficients are given by t, with t[1] being the constant term. Pol can be used if one wants t[1] to be the leading coefficient:

```
? Polrev([1,2,3])
%1 = 3*x^2 + 2*x + 1
? Pol([1,2,3])
%2 = x^2 + 2*x + 3
```

The reciprocal function of Pol (resp. Polrev) is Vec (resp. Vecrev).

Qfb(*b*, *c*, *D*, *precision*)

Creates the binary quadratic form $ax^2 + bxy + cy^2$. If $b^2 - 4ac > 0$, initialize Shanks' distance function to D. Negative definite forms are not implemented, use their positive definite counterpart instead.

Ser(v, d, serprec)

Transforms the object s into a power series with main variable v (x by default) and precision (number of significant terms) equal to $d \ge 0$ (d = series precision by default). If s is a scalar, this gives a constant power series in v with precision d. If s is a polynomial, the polynomial is truncated to d terms if needed

```
? \ps
seriesprecision = 16 significant terms
? Ser(1) \\ 16 terms by default
%1 = 1 + 0(x^16)
? Ser(1, 'y, 5)
%2 = 1 + 0(y^5)
? Ser(x^2,, 5)
%3 = x^2 + 0(x^7)
? T = polcyclo(100)
%4 = x^40 - x^30 + x^20 - x^10 + 1
? Ser(T, 'x, 11)
%5 = 1 - x^10 + 0(x^11)
```

The function is more or less equivalent with multiplication by $1 + O(v^d)$ in these cases, only faster.

For the remaining types, vectors and power series, we first explain what occurs if d is omitted. In this case, the function uses exactly the amount of information given in the input:

- If s is already a power series in v, we return it verbatim;
- If s is a vector, the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (as in Polrev(x)); in other words we convert t_VEC / t_COL to the power series whose significant terms are exactly given by the vector entries.

On the other hand, if d is explicitly given, we abide by its value and return a series, truncated or extended with zeros as needed, with d significant terms.

```
? v = [1,2,3];
? Ser(v, t) \\ 3 terms: seriesprecision is ignored!
%7 = 1 + 2*t + 3*t^2 + 0(t^3)
? Ser(v, t, 7) \\ 7 terms as explicitly requested
%8 = 1 + 2*t + 3*t^2 + 0(t^7)
? s = 1+x+0(x^2);
? Ser(s)
%10 = 1 + x + 0(x^2) \\ 2 terms: seriesprecision is ignored
? Ser(s, x, 7) \\ extend to 7 terms
%11 = 1 + x + 0(x^7)
? Ser(s, x, 1) \\ truncate to 1 term
%12 = 1 + 0(x)
```

The warning given for Pol also applies here: this is not a substitution function.

Set()

Converts x into a set, i.e. into a row vector, with strictly increasing entries with respect to the (some-what arbitrary) universal comparison function cmp. Standard container types t_VEC, t_COL, t_LIST and t_VECSMALL are converted to the set with corresponding elements. All others are converted to a set with one element.

? Set([1,2,4,2,1,3])
%1 = [1, 2, 3, 4]
? Set(x)
%2 = [x]
? Set(Vecsmall([1,3,2,1,3]))
%3 = [1, 2, 3]
Strchr()

Deprecated alias for strchr.

Vec(n)

Transforms the object x into a row vector. The dimension of the resulting vector can be optionally specified via the extra parameter n. If n is omitted or 0, the dimension depends on the type of x; the vector has a single component, except when x is

- a vector or a quadratic form: returns the initial object considered as a row vector,
- a polynomial or a power series: returns a vector consisting of the coefficients. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree. In particular the valuation is ignored (which makes the function useful for series of negative valuation):

```
? Vec(3*x<sup>2</sup> + x)
%1 = [3, 1, 0]
? Vec(x<sup>2</sup> + 3*x<sup>3</sup> + 0(x<sup>5</sup>))
%2 = [1, 3, 0]
? Vec(x<sup>2</sup> + 3*x<sup>1</sup> + 0(x))
%3 = [1, 3, 0]
```

Vec is the reciprocal function of Pol for a polynomial and of Ser for power series of valuation 0.

• a matrix: returns the vector of columns comprising the matrix,

```
? m = [1,2,3;4,5,6]
%4 =
[1 2 3]
[4 5 6]
? Vec(m)
%5 = [[1, 4]~, [2, 5]~, [3, 6]~]
```

• a character string: returns the vector of individual characters,

```
? Vec("PARI")
%6 = ["P", "A", "R", "I"]
```

- a map: returns the vector of the domain of the map,
- an error context (t_ERROR): returns the error components, see iferr.

In the last four cases (matrix, character string, map, error), n is meaningless and must be omitted or an error is raised. Otherwise, if n is given, 0 entries are appended at the end of the vector if n > 0, and prepended at the beginning if n < 0. The dimension of the resulting vector is ||n||. This allows to write a conversion function for series that takes positive valuations into account:

```
? serVec(s) = Vec(s, -serprec(s,variable(s)));
? Vec(x^2 + 3*x^3 + 0(x^5))
%2 = [0, 0, 1, 3, 0]
```

(That function is not intended for series of negative valuation.)

Vecrev(*n*)

As Vec(x, -n), then reverse the result. In particular, Vecrev is the reciprocal function of Polrev: the coefficients of the vector start with the constant coefficient of the polynomial and the others follow by increasing degree.

Vecsmall(*n*)

Transforms the object x into a row vector of type t_VECSMALL. The dimension of the resulting vector can be optionally specified via the extra parameter n.

This acts as Vec(x, n), but only on a limited set of objects: the result must be representable as a vector of small integers. If x is a character string, a vector of individual characters in ASCII encoding is returned (strchr yields back the character string).

abs(precision)

Absolute value of x (modulus if x is complex). Rational functions are not allowed. Contrary to most transcendental functions, an exact argument is *not* converted to a real number before applying **abs** and an exact result is returned if possible.

? abs(-1)
%1 = 1
? abs(3/7 + 4/7*I)
%2 = 5/7
? abs(1 + I)
%3 = 1.414213562373095048801688724

If x is a polynomial, returns -x if the leading coefficient is real and negative else returns x. For a power series, the constant coefficient is considered instead.

acos(*precision*)

Principal branch of $\cos^{-1}(x) = -i \log(x + i\sqrt{1 - x^2})$. In particular, $\Re(acos(x)) \in [0, \pi]$ and if $x \in \mathbb{R}$ and ||x|| > 1, then acos(x) is complex. The branch cut is in two pieces:] - oo, -1], continuous with quadrant II, and [1, +oo], continuous with quadrant IV. We have $acos(x) = \pi/2 - asin(x)$ for all x.

acosh(precision)

Principal branch of $\cosh^{-1}(x) = 2\log(\sqrt{(x+1)/2} + \sqrt{(x-1)/2})$. In particular, $\Re(acosh(x)) \ge 0$ and $\Im(acosh(x)) \in]-\pi,\pi]$; if $x \in \mathbb{R}$ and x < 1, then acosh(x) is complex.

addprimes()

Adds the integers contained in the vector x (or the single integer x) to a special table of "user-defined primes", and returns that table. Whenever factor is subsequently called, it will trial divide by the elements in this table. If x is empty or omitted, just returns the current list of extra primes.

The entries in x must be primes: there is no internal check, even if the factor_proven default is set. To remove primes from the list use removeprimes.

agm(y, precision)

Arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the optimal AGM is returned (the largest in absolute value over all choices of the signs of the square roots). p-adic or power series arguments are also allowed. Note that a p-adic agm exists only if x/y is congruent to 1 modulo p (modulo 16 for p = 2). x and y cannot both be vectors or matrices.

airy(precision)

Airy [Ai, Bi] functions of argument z.

```
? [A,B] = airy(1);
? A
%2 = 0.13529241631288141552414742351546630617
? B
%3 = 1.2074235949528712594363788170282869954
```

algadd(x, y)

Given two elements x and y in al, computes their sum x + y in the algebra al.

? A = alginit(nfinit(y),[-1,1]); ? algadd(A,[1,0]~,[1,2]~) %2 = [2, 2]~

Also accepts matrices with coefficients in al.

algalgtobasis(x)

Given an element x in the central simple algebra al output by alginit, transforms it to a column vector on the integral basis of al. This is the inverse function of algbasistoalg.

? A = alginit(nfinit(y²⁻⁵),[2,y]); ? algalgtobasis(A,[y,1]~) %2 = [0, 2, 0, -1, 2, 0, 0, 0]~ ? algbasistoalg(A,algalgtobasis(A,[y,1]~)) %3 = [Mod(Mod(y, y² - 5), x² - 2), 1]~

algaut()

Given a cyclic algebra $al = (L/K, \sigma, b)$ output by alginit, returns the automorphism σ .

```
? nf = nfinit(y);
? p = idealprimedec(nf,7)[1];
? p2 = idealprimedec(nf,11)[1];
? A = alginit(nf,[3,[[p,p2],[1/3,2/3]],[0]]);
? algaut(A)
%5 = -1/3*x^2 + 1/3*x + 26/3
```

algb()

Given a cyclic algebra $al = (L/K, \sigma, b)$ output by alginit, returns the element $b \in K$.

```
nf = nfinit(y);
? p = idealprimedec(nf,7)[1];
? p2 = idealprimedec(nf,11)[1];
? A = alginit(nf,[3,[[p,p2],[1/3,2/3]],[0]]);
? algb(A)
%5 = Mod(-77, y)
```

algbasis()

Given a central simple algebra *al* output by alginit, returns a \mathbb{Z} -basis of the order O_0 stored in *al* with respect to the natural order in *al*. It is a maximal order if one has been computed.

```
A = alginit(nfinit(y), [-1,-1]);
? algbasis(A)
%2 =
[1 0 0 1/2]
[0 1 0 1/2]
[0 0 1 1/2]
[0 0 0 1/2]
```

algbasistoalg(x)

Given an element *x* in the central simple algebra *al* output by alginit, transforms it to its algebraic representation in *al*. This is the inverse function of algalgtobasis.

```
? A = alginit(nfinit(y^2-5),[2,y]);
? z = algbasistoalg(A,[0,1,0,0,2,-3,0,0]~);
? liftall(z)
%3 = [(-1/2*y - 2)*x + (-1/4*y + 5/4), -3/4*y + 7/4]~
? algalgtobasis(A,z)
%4 = [0, 1, 0, 0, 2, -3, 0, 0]~
```

algcenter()

If *al* is a table algebra output by algtableinit, returns a basis of the center of the algebra *al* over its prime field (\mathbb{Q} or \mathbb{F}_p). If *al* is a central simple algebra output by alginit, returns the center of *al*, which is stored in *al*.

A simple example: the 2x2 upper triangular matrices over \mathbb{Q} , generated by I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$: the diagonal matrices form the center.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algcenter(A) \\ = (I_2)
%3 =
[1]
[0]
[0]
```

An example in the central simple case:

```
? nf = nfinit(y^3-y+1);
? A = alginit(nf, [-1,-1]);
? algcenter(A).pol
%3 = y^3 - y + 1
```

algcentralproj(z, maps)

Given a table algebra al output by algtableinit and a t_VEC $z = [z_1, ..., z_n]$ of orthogonal central idempotents, returns a t_VEC $[al_1, ..., al_n]$ of algebras such that $al_i = z_i al$. If maps = 1, each al_i is a

 t_VEC [quo, proj, lift] where quo is the quotient algebra, proj is a t_MAT representing the projection onto this quotient and lift is a t_MAT representing a lift.

A simple example: $\mathbb{F}_2 x \mathbb{F}_4$, generated by 1 = (1, 1), e = (1, 0) and x such that $x^2 + x + 1 = 0$. We have $e^2 = e$, $x^2 = x + 1$ and ex = 0.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? e = [0,1,0]~;
? e2 = algsub(A,[1,0,0]~,e);
? [a,a2] = algcentralproj(A,[e,e2]);
? algdim(a)
%6 = 1
? algdim(a2)
%7 = 2
```

algchar()

Given an algebra *al* output by alginit or algtableinit, returns the characteristic of *al*.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,13);
? algchar(A)
%3 = 13
```

algcharpoly(b, v, abs)

Given an element b in al, returns its characteristic polynomial as a polynomial in the variable v. If al is a table algebra output by algtableinit or if abs = 1, returns the absolute characteristic polynomial of b, which is an element of $\mathbb{F}_p[v]$ or $\mathbb{Q}[v]$; if al is a central simple algebra output by alginit and abs = 0, returns the reduced characteristic polynomial of b, which is an element of K[v] where K is the center of al.

```
? al = alginit(nfinit(y), [-1,-1]); \\ (-1,-1)_Q
? algcharpoly(al, [0,1]~)
%2 = x^2 + 1
? algcharpoly(al, [0,1]~,,1)
%3 = x^4 + 2*x^2 + 1
? nf = nfinit(y^2-5);
? al = alginit(nf,[-1,y]);
? a = [y,1+x]~*Mod(1,y^2-5)*Mod(1,x^2+1);
? P = lift(algcharpoly(al,a))
%7 = x^2 - 2*y*x + (-2*y + 5)
? algcharpoly(al,a,,1)
%8 = x^8 - 20*x^6 - 80*x^5 + 110*x^4 + 800*x^3 + 1500*x^2 - 400*x + 25
? lift(P*subst(P,y,-y)*Mod(1,y^2-5))^2
%9 = x^8 - 20*x^6 - 80*x^5 + 110*x^4 + 800*x^3 + 1500*x^2 - 400*x + 25
```

Also accepts a square matrix with coefficients in al.

algdegree()

Given a central simple algebra *al* output by alginit, returns the degree of *al*.

? nf = nfinit(y^3-y+1); ? A = alginit(nf, [-1,-1]); ? algdegree(A) %3 = 2

algdep(k, flag)

z being real/complex, or p-adic, finds a polynomial (in the variable 'x) of degree at most k, with integer coefficients, having z as approximate root. Note that the polynomial which is obtained is not necessarily the "correct" one. In fact it is not even guaranteed to be irreducible. One can check the closeness either by a polynomial evaluation (use subst), or by computing the roots of the polynomial given by algdep (use polroots or polrootspadic).

Internally, $lindep([1, z, ..., z^k], flag)$ is used. A nonzero value of flag may improve on the default behavior if the input number is known to a *huge* accuracy, and you suspect the last bits are incorrect: if flag > 0 the computation is done with an accuracy of flag decimal digits; to get meaningful results, the parameter flag should be smaller than the number of correct decimal digits in the input. But default values are usually sufficient, so try without flag first:

```
? \p200
? z = 2^(1/6)+3^(1/5);
? algdep(z, 30); \\ right in 280ms
? algdep(z, 30, 100); \\ wrong in 169ms
? algdep(z, 30, 170); \\ right in 288ms
? algdep(z, 30, 200); \\ wrong in 320ms
? \p250
? z = 2^(1/6)+3^(1/5); \\ recompute to new, higher, accuracy !
? algdep(z, 30); \\ right in 329ms
? algdep(z, 30, 200); \\ right in 324ms
? \p500
? algdep(2^(1/6)+3^(1/5), 30); \\ right in 677ms
? \p1000
? algdep(2^(1/6)+3^(1/5), 30); \\ right in 1.5s
```

The changes in realprecision only affect the quality of the initial approximation to $2^{1/6} + 3^{1/5}$, algdep itself uses exact operations. The size of its operands depend on the accuracy of the input of course: more accurate input means slower operations.

Proceeding by increments of 5 digits of accuracy, algdep with default flag produces its first correct result at 195 digits, and from then on a steady stream of correct results:

```
\ assume T contains the correct result, for comparison forstep(d=100, 250, 5, localprec(d);
print(d, " ", algdep(2^(1/6)+3^(1/5),30) == T))
```

The above example is the test case studied in a 2000 paper by Borwein and Lisonek: Applications of integer relation algorithms, *Discrete Math.*, **217**, p. 65–82. The version of PARI tested there was 1.39, which succeeded reliably from precision 265 on, in about 200 as much time as the current version.

algdim(abs)

If *al* is a table algebra output by algtableinit or if abs = 1, returns the dimension of *al* over its prime subfield (\mathbb{Q} or \mathbb{F}_p). If *al* is a central simple algebra output by alginit and abs = 0, returns the dimension of *al* over its center.

```
? nf = nfinit(y^3-y+1);
? A = alginit(nf, [-1,-1]);
? algdim(A)
%3 = 4
? algdim(A,1)
%4 = 12
```

algdisc()

Given a central simple algebra *al* output by alginit, computes the discriminant of the order O_0 stored in *al*, that is the determinant of the trace form $\text{Tr} : O_0 x O_0 \to \mathbb{Z}$.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-3,1-y]);
? [PR,h] = alghassef(A)
%3 = [[[2, [2, 0]~, 1, 2, 1], [3, [3, 0]~, 1, 2, 1]], Vecsmall([0, 1])]
? n = algdegree(A);
? D = algdim(A,1);
? h = vector(#h, i, n - gcd(n,h[i]));
? n^D * nf.disc^(n^2) * idealnorm(nf, idealfactorback(nf,PR,h))^n
%4 = 12960000
? algdisc(A)
%5 = 12960000
```

algdivl(x, y)

Given two elements x and y in al, computes their left quotient $x \setminus y$ in the algebra al: an element z such that xz = y (such an element is not unique when x is a zerodivisor). If x is invertible, this is the same as $x^{-1}y$. Assumes that y is left divisible by x (i.e. that z exists). Also accepts matrices with coefficients in al.

algdivr(x, y)

Given two elements x and y in al, returns xy^{-1} . Also accepts matrices with coefficients in al.

alggroup(p)

Initializes the group algebra K[G] over $K = \mathbb{Q}$ (p omitted) or \mathbb{F}_p where G is the underlying group of the galoisinit structure gal. The input gal is also allowed to be a t_VEC of permutations that is closed under products.

Example:

```
? K = nfsplitting(x^3-x+1);
? gal = galoisinit(K);
? al = alggroup(gal);
? algissemisimple(al)
%4 = 1
? G = [Vecsmall([1,2,3]), Vecsmall([1,3,2])];
? al2 = alggroup(G, 2);
? algissemisimple(al2)
%8 = 0
```

alggroupcenter(p, cc)

Initializes the center Z(K[G]) of the group algebra K[G] over $K = \mathbb{Q}$ (p = 0 or omitted) or \mathbb{F}_p where G is the underlying group of the galoisinit structure gal. The input gal is also allowed to be a t_VEC of permutations that is closed under products. Sets cc to a t_VEC [elts, conjclass, rep, flag] where elts is a sorted t_VEC containing the list of elements of G, conjclass is a t_VECSMALL of the same length as elts containing the index of the conjugacy class of the corresponding element (an integer between 1 and the number of conjugacy classes), and rep is a t_VECSMALL of length the number of conjugacy class the index in elts of a representative of this conjugacy class. Finally flag is 1 if and only if the permutation representation of G is transitive, in which case the *i*-th element of elts is characterized by g[1] = i; this is always the case when gal is a galoisinit structure. The basis of Z(K[G]) as output consists of the indicator functions of the conjugacy classes in the ordering given by cc. Example:

```
? K = nfsplitting(x^4+x+1);
? gal = galoisinit(K); \\ S4
? al = alggroupcenter(gal,,&cc);
? algiscommutative(al)
%4 = 1
? #cc[3] \\ number of conjugacy classes of S4
%5 = 5
? gal = [Vecsmall([1,2,3]),Vecsmall([1,3,2])]; \\ C2
? al = alggroupcenter(gal,,&cc);
? cc[3]
%8 = Vecsmall([1, 2])
? cc[4]
%9 = 0
```

alghasse(pl)

Given a central simple algebra al output by alginit and a prime ideal or an integer between 1 and $r_1 + r_2$, returns a t_FRAC h: the local Hasse invariant of al at the place specified by pl.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? alghasse(A, 1)
%3 = 1/2
? alghasse(A, 2)
%4 = 0
? alghasse(A, idealprimedec(nf,2)[1])
%5 = 1/2
? alghasse(A, idealprimedec(nf,5)[1])
%6 = 0
```

alghassef()

Given a central simple algebra al output by alginit, returns a t_VEC $[PR, h_f]$ describing the local Hasse invariants at the finite places of the center: PR is a t_VEC of primes and h_f is a t_VECSMALL of integers modulo the degree d of al. The Hasse invariant of al at the primes outside PR is 0, but PR can include primes at which the Hasse invariant is 0.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,2*y-1]);
? [PR,hf] = alghassef(A);
? PR
%4 = [[19, [10, 2]~, 1, 1, [-8, 2; 2, -10]], [2, [2, 0]~, 1, 2, 1]]
? hf
%5 = Vecsmall([1, 0])
```

alghassei()

Given a central simple algebra al output by alginit, returns a t_VECSMALL h_i of r_1 integers modulo the degree d of al, where r_1 is the number of real places of the center: the local Hasse invariants of al at infinite places.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? alghassei(A)
%3 = Vecsmall([1, 0])
```

algindex(pl)

Returns the index of the central simple algebra A over K (as output by alginit), that is the degree e of the unique central division algebra D over K such that A is isomorphic to some matrix algebra $M_k(D)$. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case return the local index at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algindex(A, 1)
%3 = 2
? algindex(A, 2)
%4 = 1
? algindex(A, idealprimedec(nf,2)[1])
%5 = 2
? algindex(A, idealprimedec(nf,5)[1])
%6 = 1
? algindex(A)
%7 = 2
```

alginit(C, v, maxord)

Initializes the central simple algebra defined by data B, C and variable v, as follows.

• (multiplication table) B is the base number field K in nfinit form, C is a "multiplication table" over K. As a K-vector space, the algebra is generated by a basis $(e_1 = 1, ..., e_n)$; the table is given as a t_VEC of n matrices in $M_n(K)$, giving the left multiplication by the basis elements e_i , in the given basis. Assumes that $e_1 = 1$, that the multiplication table is integral, and that $(\bigoplus_{i=1}^n Ke_i, C)$ describes a central simple algebra over K.

```
{ mi = [0,-1,0, 0;
1, 0,0, 0;
0, 0,0,-1;
0, 0,1, 0];
mj = [0, 0,-1,0;
0, 0, 0,1;
1, 0, 0,0;
0,-1, 0,0];
mk = [0, 0, 0, 0;
0, 0,-1, 0;
0, 1, 0, 0;
1, 0, 0,-1];
A = alginit(nfinit(y), [matid(4), mi,mj,mk], 0); }
```

represents (in a complicated way) the quaternion algebra $(-1, -1)_{\mathbb{Q}}$. See below for a simpler solution.

• (cyclic algebra) *B* is an rnf structure attached to a cyclic number field extension L/K of degree *d*, *C* is a t_VEC [sigma,b] with 2 components: sigma is a t_POLMOD representing an automorphism generating Gal(L/K), *b* is an element in K^* . This represents the cyclic algebra $(L/K, \sigma, b)$. Currently the element *b* has to be integral.

```
? Q = nfinit(y); T = polcyclo(5, 'x); F = rnfinit(Q, T);
? A = alginit(F, [Mod(x^2,T), 3]);
```

defines the cyclic algebra $(L/\mathbb{Q}, \sigma, 3)$, where $L = \mathbb{Q}(\zeta_5)$ and $\sigma : \zeta : --- > \zeta^2$ generates $Gal(L/\mathbb{Q})$.

• (quaternion algebra, special case of the above) B is an nf structure attached to a number field K, C = [a, b] is a vector containing two elements of K^* with a not a square in K, returns the quaternion algebra $(a, b)_K$. The variable v ('x by default) must have higher priority than the variable of K.pol and is used to represent elements in the splitting field $L = K[x]/(x^2 - a)$.

```
? Q = nfinit(y); A = alginit(Q, [-1,-1]); \\ (-1,-1)_Q
```

• (algebra/K defined by local Hasse invariants) B is an nf structure attached to a number field $K, C = [d, [PR, h_f], h_i]$ is a triple containing an integer d > 1, a pair $[PR, h_f]$ describing the Hasse invariants at finite places, and h_i the Hasse invariants at archimedean (real) places. A local Hasse invariant belongs to $(1/d)\mathbb{Z}/\mathbb{Z} \subset \mathbb{Q}/\mathbb{Z}$, and is given either as a t_FRAC (lift to $(1/d)\mathbb{Z})$, a t_INT or t_INTMOD modulo d (lift to $\mathbb{Z}/d\mathbb{Z}$); a whole vector of local invariants can also be given as a t_VECSMALL, whose entries are handled as t_INT s. PR is a list of prime ideals (prid structures), and h_f is a vector of the same length giving the local invariants at those maximal ideals. The invariants at infinite real places are indexed by the real roots K.roots: if the Archimedean place v is attached to the j-th root, the value of h_v is given by $h_i[j]$, must be 0 or 1/2 (or d/2 modulo d), and can be nonzero only if d is even.

By class field theory, provided the local invariants h_v sum to 0, up to Brauer equivalence, there is a unique central simple algebra over K with given local invariants and trivial invariant elsewhere. In particular, up to isomorphism, there is a unique such algebra A of degree d.

We realize A as a cyclic algebra through class field theory. The variable v ('x by default) must have higher priority than the variable of K.pol and is used to represent elements in the (cyclic) splitting field extension L/K for A.

```
? nf = nfinit(y<sup>2</sup>+1);
? PR = idealprimedec(nf,5); #PR
%2 = 2
? hi = [];
? hf = [PR, [1/3,-1/3]];
? A = alginit(nf, [3,hf,hi]);
? algsplittingfield(A).pol
%6 = x<sup>3</sup> - 21*x + 7
```

• (matrix algebra, toy example) B is an nf structure attached to a number field K, C = d is a positive integer. Returns a cyclic algebra isomorphic to the matrix algebra $M_d(K)$.

In all cases, this function computes a maximal order for the algebra by default, which may require a lot of time. Setting maxord = 0 prevents this computation.

The pari object representing such an algebra A is a t_VEC with the following data:

- A splitting field L of A of the same degree over K as A, in rnfinit format, accessed with algsplittingfield.
- The Hasse invariants at the real places of K, accessed with alghassei.
- The Hasse invariants of A at the finite primes of K that ramify in the natural order of A, accessed with alghassef.
- A basis of an order O_0 expressed on the basis of the natural order, accessed with algebrasis.
- A basis of the natural order expressed on the basis of O_0 , accessed with alginvbasis.
- The left multiplication table of O_0 on the previous basis, accessed with algmultable.
- The characteristic of A (always 0), accessed with algchar.
- The absolute traces of the elements of the basis of O_0 .

- If A was constructed as a cyclic algebra $(L/K, \sigma, b)$ of degree d, a t_VEC $[\sigma, \sigma^2, ..., \sigma^{d-1}]$. The function algaut returns σ .
- If A was constructed as a cyclic algebra $(L/K, \sigma, b)$, the element b, accessed with algb.
- If A was constructed with its multiplication table mt over K, the t_VEC of t_MAT mt, accessed with algrelmultable.
- If A was constructed with its multiplication table mt over K, a t_VEC with three components: a t_COL representing an element of A generating the splitting field L as a maximal subfield of A, a t_MAT representing an L-basis B of A expressed on the \mathbb{Z} -basis of O_0 , and a t_MAT representing the \mathbb{Z} -basis of O_0 expressed on B. This data is accessed with algsplittingdata.

alginv(x)

Given an element x in al, computes its inverse x^{-1} in the algebra al. Assumes that x is invertible.

```
? A = alginit(nfinit(y), [-1,-1]);
? alginv(A,[1,1,0,0]~)
%2 = [1/2, 1/2, 0, 0]~
```

Also accepts matrices with coefficients in *al*.

alginvbasis()

Given an central simple algebra *al* output by alginit, returns a \mathbb{Z} -basis of the natural order in *al* with respect to the order O_0 stored in *al*.

```
A = alginit(nfinit(y), [-1,-1]);
? alginvbasis(A)
%2 =
[1 0 0 -1]
[0 1 0 -1]
[0 0 1 -1]
[0 0 0 2]
```

algisassociative(p)

Returns 1 if the multiplication table mt is suitable for algtableinit(mt,p), 0 otherwise. More precisely, mt should be a t_VEC of n matrices in $M_n(K)$, giving the left multiplications by the basis elements $e_1, ..., e_n$ (structure constants). We check whether the first basis element e_1 is 1 and $e_i(e_je_k) = (e_ie_j)e_k$ for all i, j, k.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? algisassociative(mt)
%2 = 1
```

May be used to check a posteriori an algebra: we also allow mt as output by algtableinit (p is ignored in this case).

algiscommutative()

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is commutative.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
```

```
? algiscommutative(A)
%3 = 0
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? algiscommutative(A)
%6 = 1
```

algisdivision(pl)

Given a central simple algebra al output by alginit, tests whether al is a division algebra. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally a division algebra at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algisdivision(A, 1)
%3 = 1
? algisdivision(A, 2)
%4 = 0
? algisdivision(A, idealprimedec(nf,2)[1])
%5 = 1
? algisdivision(A, idealprimedec(nf,5)[1])
%6 = 0
? algisdivision(A)
%7 = 1
```

algisdivl(x, y, z)

Given two elements x and y in al, tests whether y is left divisible by x, that is whether there exists z in al such that xz = y, and sets z to this element if it exists.

```
? A = alginit(nfinit(y), [-1,1]);
? algisdivl(A,[x+2,-x-2]~,[x,1]~)
%2 = 0
? algisdivl(A,[x+2,-x-2]~,[-x,x]~,&z)
%3 = 1
? z
%4 = [Mod(-2/5*x - 1/5, x^2 + 1), 0]~
```

Also accepts matrices with coefficients in al.

algisinv(x, ix)

Given an element x in al, tests whether x is invertible, and sets ix to the inverse of x.

```
? A = alginit(nfinit(y), [-1,1]);
? algisinv(A,[-1,1]~)
%2 = 0
? algisinv(A,[1,2]~,&ix)
%3 = 1
? ix
%4 = [Mod(Mod(-1/3, y), x^2 + 1), Mod(Mod(2/3, y), x^2 + 1)]~
```

Also accepts matrices with coefficients in al.

algisramified(pl)

Given a central simple algebra *al* output by alginit, tests whether *al* is ramified, i.e. not isomorphic to a

matrix algebra over its center. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally ramified at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algisramified(A, 1)
%3 = 1
? algisramified(A, 2)
%4 = 0
? algisramified(A, idealprimedec(nf,2)[1])
%5 = 1
? algisramified(A, idealprimedec(nf,5)[1])
%6 = 0
? algisramified(A)
%7 = 1
```

algissemisimple()

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is semisimple.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algissemisimple(A)
%3 = 0
? m_i=[0,-1,0,0;1,0,0,0;0,0,0,-1;0,0,1,0]; \\ quaternion algebra (-1,-1)
? m_j=[0,0,-1,0;0,0,0,1;1,0,0,0;0,-1,0,0];
? m_k=[0,0,0,-1;0,0,-1,0;0,1,0,0;1,0,0,0];
? mt = [matid(4), m_i, m_j, m_k];
? A = algtableinit(mt);
? algissemisimple(A)
%9 = 1
```

algissimple(ss)

al being a table algebra output by algtableinit or a central simple algebra output by alginit, tests whether the algebra *al* is simple. If ss = 1, assumes that the algebra *al* is semisimple without testing it.

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt); \\ matrices [*,*; 0,*]
? algissimple(A)
\%3 = 0
? algissimple(A,1) \\ incorrectly assume that A is semisimple
\%4 = 1
? m_i=[0,-1,0,0;1,0,0,0;0,0,0,-1;0,0,1,0];
? m_j=[0,0,-1,0;0,0,0,1;1,0,0,0;0,-1,0,0];
? m_k=[0,0,0,-1;0,0,b,0;0,1,0,0;1,0,0,0];
? mt = [matid(4), m_i, m_j, m_k];
? A = algtableinit(mt); \\ quaternion algebra (-1,-1)
? algissimple(A)
%10 = 1
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2); \\ direct product F_4 x F_2
? algissimple(A)
%13 = 0
```

algissplit(pl)

Given a central simple algebra al output by alginit, tests whether al is split, i.e. isomorphic to a matrix algebra over its center. If pl is set, it should be a prime ideal of K or an integer between 1 and $r_1 + r_2$, and in that case tests whether al is locally split at the place pl instead.

```
? nf = nfinit(y^2-5);
? A = alginit(nf, [-1,y]);
? algissplit(A, 1)
%3 = 0
? algissplit(A, 2)
%4 = 1
? algissplit(A, idealprimedec(nf,2)[1])
%5 = 0
? algissplit(A, idealprimedec(nf,5)[1])
%6 = 1
? algissplit(A)
%7 = 0
```

alglatadd(lat1, lat2, ptinter)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the sum *lat1* + *lat2*. If *ptinter* is present, set it to the intersection *lat1* \cap *lat2*.

? al = alginit(nfinit(y²+7), [-1,-1]); ? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~); ? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~); ? latsum = alglatadd(al,lat1,lat2,&latinter); ? matdet(latsum[1]) %5 = 4 ? matdet(latinter[1]) %6 = 64

alglatcontains(lat, x, ptc)

Given an algebra al, a lattice *lat* and x in al, tests whether $x \in lat$. If *ptc* is present, sets it to the t_COL of coordinates of x in the basis of *lat*.

```
? al = alginit(nfinit(y<sup>2</sup>+7), [-1,-1]);
? a1 = [1,-1,0,1,2,0,1,2]~;
? lat1 = alglathnf(al,a1);
? alglatcontains(al,lat1,a1,&c)
%4 = 1
? c
%5 = [-1, -2, -1, 1, 2, 0, 1, 1]~
```

alglatelement(lat, c)

Given an algebra *al*, a lattice *lat* and a $t_COL c$, returns the element of *al* whose coordinates on the mathbb{Z}-basis of *lat* are given by *c*.

```
? al = alginit(nfinit(y<sup>2</sup>+7), [-1,-1]);
? a1 = [1,-1,0,1,2,0,1,2]~;
? lat1 = alglathnf(al,a1);
? c = [1..8]~;
? elt = alglatelement(al,lat1,c);
? alglatcontains(al,lat1,elt,&c2)
```

%6 = 1		
? c==c2		
%7 = 1		

alglathnf(m, d)

Given an algebra al and a matrix m with columns representing elements of al, returns the lattice L generated by the columns of m. If provided, d must be a rational number such that L contains d times the natural basis of al. The argument m is also allowed to be a t_VEC of t_MAT, in which case m is replaced by the concatenation of the matrices, or a t_COL, in which case m is replaced by its left multiplication table as an element of al.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? a = [1,1,-1/2,1,1/3,-1,1,1]~;
? mt = algtomatrix(al,a,1);
? lat = alglathnf(al,mt);
? lat[2]
%5 = 1/6
```

alglatindex(lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the generalized index of *lat1* relative to *lat2*, i.e. $||lat2/lat1 \cap lat2||/||lat1/lat1 \cap lat2||$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? alglatindex(al,lat1,lat2)
%4 = 1
? lat1==lat2
%5 = 0
```

alglatinter(lat1, lat2, ptsum)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the intersection $lat1 \cap lat2$. If *ptsum* is present, sets it to the sum lat1 + lat2.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? latinter = alglatinter(al,lat1,lat2,&latsum);
? matdet(latsum[1])
%5 = 4
? matdet(latinter[1])
%6 = 64
```

alglatlefttransporter(lat1, lat2)

Given an algebra al and two lattices lat1 and lat2 in al, computes the left transporter from lat1 to lat2, i.e. the set of $x \in al$ such that $x.lat1 \subset lat2$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,-1,0,1,2,0,5,2]~);
? lat2 = alglathnf(al,[0,1,-2,-1,0,0,3,1]~);
? tr = alglatlefttransporter(al,lat1,lat2);
? a = alglatelement(al,tr,[0,0,0,0,0,0,1,0]~);
```

```
? alglatsubset(al,alglatmul(al,a,lat1),lat2)
%6 = 1
? alglatsubset(al,alglatmul(al,lat1,a),lat2)
%7 = 0
```

alglatmul(lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the lattice generated by the products of elements of *lat1* and *lat2*. One of *lat1* and *lat2* is also allowed to be an element of *al*; in this case, computes the product of the element and the lattice.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? a1 = [1,-1,0,1,2,0,1,2]~;
? a2 = [0,1,2,-1,0,0,3,1]~;
? lat1 = alglathnf(al,a1);
? lat2 = alglathnf(al,a2);
? lat3 = alglatmul(al,lat1,lat2);
? matdet(lat3[1])
%7 = 29584
? lat3 == alglathnf(al, algmul(al,a1,a2))
%8 = 0
? lat3 == alglatmul(al, lat1, a2)
%9 = 0
? lat3 == alglatmul(al, a1, lat2)
%10 = 0
```

alglatrighttransporter(lat1, lat2)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, computes the right transporter from *lat1* to *lat2*, i.e. the set of $x \in al$ such that $lat1.x \subset lat2$.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,matdiagonal([1,3,7,1,2,8,5,2]));
? lat2 = alglathnf(al,matdiagonal([5,3,8,1,9,8,7,1]));
? tr = alglatrighttransporter(al,lat1,lat2);
? a = alglatelement(al,tr,[0,0,0,0,0,0,0,0,1]~);
? alglatsubset(al,alglatmul(al,lat1,a),lat2)
%6 = 1
? alglatsubset(al,alglatmul(al,a,lat1),lat2)
%7 = 0
```

alglatsubset(lat1, lat2, ptindex)

Given an algebra *al* and two lattices *lat1* and *lat2* in *al*, tests whether $lat1 \subset lat2$. If it is true and *ptindex* is present, sets it to the index of *lat1* in *lat2*.

```
? al = alginit(nfinit(y^2+7), [-1,-1]);
? lat1 = alglathnf(al,[1,1,0,0,0,0,0,0]~);
? lat2 = alglathnf(al,[1,0,1,0,0,0,0,0]~);
? alglatsubset(al,lat1,lat2)
%4 = 0
? latsum = alglatadd(al,lat1,lat2);
? alglatsubset(al,lat1,latsum,&index)
%6 = 1
```

? index			
%7 = 4			

algmakeintegral(maps)

mt being a multiplication table over \mathbb{Q} in the same format as the input of algtableinit, computes an integral multiplication table *mt2* for an isomorphic algebra. When maps = 1, returns a t_VEC [mt2, S, T] where S and T are matrices respectively representing the map from the algebra defined by *mt* to the one defined by *mt2* and its inverse.

```
? mt = [matid(2),[0,-1/4;1,0]];
? algtableinit(mt);
 *** at top-level: algtableinit(mt)
 *** ^_____
 *** algtableinit: domain error in algtableinit: denominator(mt) != 1
? mt2 = algmakeintegral(mt);
? al = algtableinit(mt2);
? algisassociative(al)
%4 = 1
? [mt2, S, T] = algmakeintegral(mt,1);
? S
%6 =
[1 0]
[0 \ 1/4]
? T
%7 =
[1 0]
[0 4]
? vector(#mt, i, S * (mt * T[,i]) * T) == mt2
\%8 = 1
```

algmul(x, y)

Given two elements x and y in al, computes their product xy in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algmul(A,[1,1,0,0]~,[0,0,2,1]~) %2 = [2, 3, 5, -4]~

Also accepts matrices with coefficients in *al*.

algmultable()

Returns a multiplication table of *al* over its prime subfield (\mathbb{Q} or \mathbb{F}_p), as a t_VEC of t_MAT: the left multiplication tables of basis elements. If *al* was output by algtableinit, returns the multiplication table used to define *al*. If *al* was output by alginit, returns the multiplication table of the order O_0 stored in *al*.

```
? A = alginit(nfinit(y), [-1,-1]);
? M = algmultable(A);
? #M
%3 = 4
? M[1] \\ multiplication by e_1 = 1
%4 =
```

[1 0 0 0] [0 1 0 0] [0 0 1 0] [0 0 1 0] [0 0 0 1] ? M[2] %5 = [0 -1 1 0] [1 0 1 1] [0 0 1 1] [0 0 -2 -1]

algneg(x)

Given an element x in al, computes its opposite -x in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algneg(A,[1,1,0,0]~) %2 = [-1, -1, 0, 0]~

Also accepts matrices with coefficients in al.

algnorm(x, abs)

Given an element x in al, computes its norm. If al is a table algebra output by algtableinit or if abs = 1, returns the absolute norm of x, which is an element of \mathbb{F}_p of \mathbb{Q} ; if al is a central simple algebra output by alginit and abs = 0 (default), returns the reduced norm of x, which is an element of the center of al.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,19);
? algnorm(A,[0,-2,3]~)
%3 = 18
? nf = nfinit(y^2-5);
? B = alginit(nf,[-1,y]);
? b = [x,1]~;
? n = algnorm(B,b)
%7 = Mod(-y + 1, y^2 - 5)
? algnorm(B,b,1)
%8 = 16
? nfeltnorm(nf,n)^algdegree(B)
%9 = 16
```

Also accepts a square matrix with coefficients in al.

algpoleval(T, b)

Given an element b in al and a polynomial T in K[X], computes T(b) in al. Also accepts as input a t_VEC [b, mb] where mb is the left multiplication table of b.

? nf = nfinit(y^2-5); ? al = alginit(nf,[y,-1]); ? b = [1..8]~; ? pol = algcharpoly(al,b,,1); ? algpoleval(al,pol,b)==0 %5 = 1 ? mb = algtomatrix(al,b,1); ? algpoleval(al,pol,[b,mb])==0 %7 = 1

algpow(x, n)

Given an element x in al and an integer n, computes the power x^n in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algpow(A,[1,1,0,0]~,7) %2 = [8, -8, 0, 0]~

Also accepts a square matrix with coefficients in al.

algprimesubalg()

al being the output of algtableinit representing a semisimple algebra of positive characteristic, returns a basis of the prime subalgebra of *al*. The prime subalgebra of *al* is the subalgebra fixed by the Frobenius automorphism of the center of *al*. It is abstractly isomorphic to a product of copies of \mathbb{F}_p .

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? algprimesubalg(A)
%3 =
[1 0]
[0 1]
[0 0]
```

algquotient(I, maps)

al being a table algebra output by algtableinit and *I* being a basis of a two-sided ideal of *al* represented by a matrix, returns the quotient al/I. When maps = 1, returns a t_VEC [al/I, proj, lift] where *proj* and *lift* are matrices respectively representing the projection map and a section of it.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? AQ = algquotient(A,[0;1;0]);
? algdim(AQ)
%4 = 2
```

algradical()

al being a table algebra output by algtableinit, returns a basis of the Jacobson radical of the algebra *al* over its prime field (\mathbb{Q} or \mathbb{F}_p).

Here is an example with $A = \mathbb{Q}[x]/(x^2)$, with the basis (1, x):

```
? mt = [matid(2),[0,0;1,0]];
? A = algtableinit(mt);
```

```
? algradical(A) \\ = (x)
%3 =
[0]
[1]
```

Another one with $2x^2$ upper triangular matrices over \mathbb{Q} , with basis I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$:

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algradical(A) \\ = (a)
%6 =
[0]
[1]
[0]
```

algramifiedplaces()

Given a central simple algebra al output by alginit, returns a t_VEC containing the list of places of the center of al that are ramified in al. Each place is described as an integer between 1 and r_1 or as a prime ideal.

? nf = nfinit(y^2-5); ? A = alginit(nf, [-1,y]); ? algramifiedplaces(A) %3 = [1, [2, [2, 0]~, 1, 2, 1]]

algrandom(b)

Given an algebra *al* and an integer *b*, returns a random element in *al* with coefficients in [-b, b].

algrelmultable()

Given a central simple algebra *al* output by alginit defined by a multiplication table over its center (a number field), returns this multiplication table.

```
? nf = nfinit(y^3-5); a = y; b = y^2;
? \{m_i = [0,a,0,0;
1,0,0,0;
0,0,0,a;
0, 0, 1, 0]; \}
? \{m_j = [0, 0, b, 0;
0, 0,0,-b;
1, 0,0, 0;
0, -1, 0, 0]; \}
? \{m_k = [0, 0, 0, -a^*b;
0, 0,b, 0;
0,-a,0, 0;
1, 0, 0, 0];
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf,mt,'x);
? M = algrelmultable(A);
```

? M[2] == m_i %8 = 1 ? M[3] == m_j %9 = 1 ? M[4] == m_k %10 = 1

algsimpledec(maps)

al being the output of algtableinit, returns a t_VEC $[J, [al_1, al_2, ..., al_n]]$ where J is a basis of the Jacobson radical of al and al/J is isomorphic to the direct product of the simple algebras al_i . When maps = 1, each al_i is replaced with a t_VEC $[al_i, proj_i, lift_i]$ where $proj_i$ and $lift_i$ are matrices respectively representing the projection map $al \rightarrow al_i$ and a section of it. Modulo J, the images of the $lift_i$ form a direct sum in al/J, so that the images of 1_i under $lift_i$ are central primitive idempotents of al/J. The factors are sorted by increasing dimension, then increasing dimension of the center. This ensures that the ordering of the isomorphism classes of the factors is deterministic over finite fields, but not necessarily over \mathbb{Q} .

algsplit(v)

If *al* is a table algebra over \mathbb{F}_p output by algtableinit that represents a simple algebra, computes an isomorphism between *al* and a matrix algebra $M_d(\mathbb{F}_{p^n})$ where $N = nd^2$ is the dimension of *al*. Returns a t_VEC [map, mapi], where:

- *map* is a t_VEC of N matrices of size dxd with t_FFELT coefficients using the variable v, representing the image of the basis of al under the isomorphism.
- mapi is an NxN matrix with t_INT coefficients, representing the image in al by the inverse isomorphism of the basis (b_i) of M_d(F_p[α]) (where α has degree n over F_p) defined as follows: let E_{i,j} be the matrix having all coefficients 0 except the (i, j)-th coefficient equal to 1, and define

$$b_{i_3+n(i_2+di_1)+1} = E_{i_1+1,i_2+1}\alpha^{i_3},$$

where : math : '0 <= $i_1, i_2 < d$ 'and : math : '0 <= $i_3 < n$ '.

Example:

```
? al0 = alginit(nfinit(y^{2+7}), [-1,-1]);
? al = algtableinit(algmultable(al0), 3); \\ isomorphic to M_2(F_9)
? [map,mapi] = algsplit(al, 'a);
? x = [1,2,1,0,0,0,0]~; fx = map*x
%4 =
[2*a 0]
[ 0 2]
? y = [0,0,0,0,1,0,0,1]~; fy = map*y
%5 =
[1 2*a]
[2 a + 2]
? map*algmul(al,x,y) == fx*fy
\%6 = 1
? map*mapi[,6]
%7 =
[0 0]
[a 0]
```

Warning. If *al* is not simple, algsplit(al) can trigger an error, but can also run into an infinite loop. Example:

```
? al = alginit(nfinit(y),[-1,-1]); \\ ramified at 2
? al2 = algtableinit(algmultable(al),2); \\ maximal order modulo 2
? algsplit(al2); \\ not semisimple, infinite loop
```

algsplittingdata()

Given a central simple algebra al output by alginit defined by a multiplication table over its center K (a number field), returns data stored to compute a splitting of al over an extension. This data is a t_VEC [t,Lbas,Lbasinv] with 3 components:

- an element t of al such that L = K(t) is a maximal subfield of al;
- a matrix Lbas expressing a *L*-basis of *al* (given an *L*-vector space structure by multiplication on the right) on the integral basis of *al*;
- a matrix Lbasinv expressing the integral basis of *al* on the previous *L*-basis.

```
? nf = nfinit(y^{3}-5); a = y; b = y^{2};
? \{m_i = [0,a,0,0;
1,0,0,0;
0,0,0,a;
0, 0, 1, 0]; \}
? {m_j = [0, 0,b, 0;
0, 0,0,-b;
1, 0,0, 0;
0, -1, 0, 0]; \}
? \{m_k = [0, 0, 0, -a^*b;
0. 0.b. 0:
0,-a,0, 0;
1, 0, 0, 0];
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf, mt, 'x);
? [t,Lb,Lbi] = algsplittingdata(A);
? t
\%8 = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]~;
? matsize(Lb)
\%9 = [12, 2]
? matsize(Lbi)
\%10 = [2, 12]
```

algsplittingfield()

Given a central simple algebra *al* output by alginit, returns an rnf structure: the splitting field of *al* that is stored in *al*, as a relative extension of the center.

nf = nfinit(y³⁻⁵); a = y; b = y²; {m_i = [0,a,0,0; 1,0,0,0; 0,0,0,a; 0,0,1,0];} {m_j = [0, 0,b, 0; 0, 0,0,-b; 1, 0,0, 0;

```
0,-1,0, 0];}
{m_k = [0, 0,0,-a*b;
0, 0,b, 0;
0,-a,0, 0;
1, 0,0, 0];}
mt = [matid(4), m_i, m_j, m_k];
A = alginit(nf,mt,'x);
algsplittingfield(A).pol
%8 = x^2 - y
```

algsqr(x)

Given an element x in al, computes its square x^2 in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algsqr(A,[1,0,2,0]~) %2 = [-3, 0, 4, 0]~

Also accepts a square matrix with coefficients in al.

algsub(x, y)

Given two elements x and y in al, computes their difference x - y in the algebra al.

? A = alginit(nfinit(y), [-1,-1]); ? algsub(A,[1,1,0,0]~,[1,0,1,0]~) %2 = [0, 1, -1, 0]~

Also accepts matrices with coefficients in al.

algsubalg(B)

al being a table algebra output by algtableinit and *B* being a basis of a subalgebra of *al* represented by a matrix, computes an algebra *al2* isomorphic to *B*.

Returns [al2, B2] where B2 is a possibly different basis of the subalgebra al2, with respect to which the multiplication table of al2 is defined.

```
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? B = algsubalg(A, [1,0; 0,0; 0,1]);
? algdim(A)
\%4 = 3
? algdim(B[1])
%5 = 2
? m = matcompanion(x^{4+1});
? mt = [m^i | i <- [0..3]];
? al = algtableinit(mt);
? B = [1,0;0,0;0,1/2;0,0];
? al2 = algsubalg(al,B);
? algdim(al2[1])
? al2[2]
%13 =
[1 0]
[0 0]
```

[0 1]			
[0 0]			

algtableinit(p)

Initializes the associative algebra over $K = \mathbb{Q}$ (p omitted) or \mathbb{F}_p defined by the multiplication table *mt*. As a K-vector space, the algebra is generated by a basis $(e_1 = 1, e_2, ..., e_n)$; the table is given as a t_VEC of n matrices in $M_n(K)$, giving the left multiplication by the basis elements e_i , in the given basis. Assumes that $e_1 = 1$, that $Ke_1 \oplus ... \oplus Ke_n$] describes an associative algebra over K, and in the case $K = \mathbb{Q}$ that the multiplication table is integral. If the algebra is already known to be central and simple, then the case $K = \mathbb{F}_p$ is useless, and one should use alginit directly.

The point of this function is to input a finite dimensional K-algebra, so as to later compute its radical, then to split the quotient algebra as a product of simple algebras over K.

The pari object representing such an algebra A is a t_VEC with the following data:

- The characteristic of A, accessed with algchar.
- The multiplication table of A, accessed with algmultable.
- The traces of the elements of the basis.

A simple example: the $2x^2$ upper triangular matrices over \mathbb{Q} , generated by I_2 , a = [0, 1; 0, 0] and b = [0, 0; 0, 1], such that $a^2 = 0$, ab = a, ba = 0, $b^2 = b$:

```
? mt = [matid(3),[0,0,0;1,0,1;0,0,0],[0,0,0;0,0,0;1,0,1]];
? A = algtableinit(mt);
? algradical(A) \\ = (a)
%6 =
[0]
[1]
[0]
[0]
[0]
[0]
[0]
```

algtensor(al2, maxord)

Given two algebras *al1* and *al2*, computes their tensor product. Computes a maximal order by default. Prevent this computation by setting maxord = 0.

Currently only implemented for cyclic algebras of coprime degree over the same center K, and the tensor product is over K.

algtomatrix(x, abs)

Given an element x in al, returns the image of x under a homomorphism to a matrix algebra. If al is a table algebra output by algtableinit or if abs = 1, returns the left multiplication table on the integral basis; if al is a central simple algebra and abs = 0, returns $\phi(x)$ where $\phi : A \otimes_K L \to M_d(L)$ (where d is the

degree of the algebra and L is an extension of L with [L : K] = d) is an isomorphism stored in al. Also accepts a square matrix with coefficients in al.

```
? A = alginit(nfinit(y), [-1, -1]);
? algtomatrix(A, [0,0,0,2]~)
%2 =
[Mod(x + 1, x^2 + 1) Mod(Mod(1, y)^*x + Mod(-1, y), x^2 + 1)]
[Mod(x + 1, x^2 + 1) Mod(-x + 1, x^2 + 1)]
? algtomatrix(A,[0,1,0,0]~,1)
%2 =
[0 -1 1 0]
[1 0 1 1]
[0 \ 0 \ 1 \ 1]
[0 \ 0 \ -2 \ -1]
? algtomatrix(A,[0,x]~,1)
%3 =
[-1 \ 0 \ 0 \ -1]
[-1 \ 0 \ 1 \ 0]
[-1 -1 \ 0 \ -1]
[2001]
```

Also accepts matrices with coefficients in al.

algtrace(x, abs)

Given an element x in al, computes its trace. If al is a table algebra output by algtableinit or if abs = 1, returns the absolute trace of x, which is an element of \mathbb{F}_p or \mathbb{Q} ; if al is the output of alginit and abs = 0 (default), returns the reduced trace of x, which is an element of the center of al.

```
? A = alginit(nfinit(y), [-1,-1]);
? algtrace(A,[5,0,0,1]~)
%2 = 11
? algtrace(A,[5,0,0,1]~,1)
%3 = 22
? nf = nfinit(y^2-5);
? A = alginit(nf,[-1,y]);
? a = [1+x+y,2*y]~*Mod(1,y^2-5)*Mod(1,x^2+1);
? t = algtrace(A,a)
%7 = Mod(2*y + 2, y^2 - 5)
? algtrace(A,a,1)
%8 = 8
? algdegree(A)*nfelttrace(nf,t)
%9 = 8
```

Also accepts a square matrix with coefficients in *al*.

algtype()

Given an algebra *al* output by alginit or by algtableinit, returns an integer indicating the type of algebra:

- 0: not a valid algebra.
- 1: table algebra output by algtableinit.
- 2: central simple algebra output by alginit and represented by a multiplication table over its center.
- 3: central simple algebra output by alginit and represented by a cyclic algebra.

```
? algtype([])
%1 = 0
? mt = [matid(3), [0,0,0; 1,1,0; 0,0,0], [0,0,1; 0,0,0; 1,0,1]];
? A = algtableinit(mt,2);
? algtype(A)
\%4 = 1
? nf = nfinit(y^3-5);
? a = y; b = y^2;
? \{m_i = [0,a,0,0;
1,0,0,0;
0,0,0,a;
 0, 0, 1, 0]; \}
? {m_j = [0, 0, b, 0;}
 0, 0,0,-b;
 1, 0,0, 0;
0, -1, 0, 0]; \}
? \{m_k = [0, 0, 0, -a^*b;
 0, 0,b, 0;
0,-a,0, 0;
1, 0, 0, 0];
? mt = [matid(4), m_i, m_j, m_k];
? A = alginit(nf,mt,'x);
? algtype(A)
%12 = 2
? A = alginit(nfinit(y), [-1,-1]);
? algtype(A)
\%14 = 3
```

apply(A)

Apply the t_CLOSURE f to the entries of A.

- If A is a scalar, return f(A).
- If A is a polynomial or power series $\sum a_i x^i$ (+O(x^N)), apply f on all coefficients and return $\sum f(a_i)x^i$ (+O(x^N)).
- If A is a vector or list $[a_1, ..., a_n]$, return the vector or list $[f(a_1), ..., f(a_n)]$. If A is a matrix, return the matrix whose entries are the f(A[i, j]).

```
? apply(x->x<sup>2</sup>, [1,2,3,4])
%1 = [1, 4, 9, 16]
? apply(x->x<sup>2</sup>, [1,2;3,4])
%2 =
[1 4]
[9 16]
? apply(x->x<sup>2</sup>, 4*x<sup>2</sup> + 3*x+ 2)
%3 = 16*x<sup>2</sup> + 9*x + 4
```

? apply(sign, $2 - 3^* x + 4^*x^2 + 0(x^3)$) %4 = 1 - x + x^2 + 0(x^3)

Note that many functions already act componentwise on vectors or matrices, but they almost never act on lists; in this case, apply is a good solution:

```
? L = List([Mod(1,3), Mod(2,4)]);
? lift(L)
*** at top-level: lift(L)
*** ^-----
*** lift: incorrect type in lift.
? apply(lift, L);
%2 = List([1, 2])
```

Remark. For v a t_VEC, t_COL, t_VECSMALL, t_LIST or t_MAT, the alternative set-notations

```
[g(x) | x <- v, f(x)]
[x | x <- v, f(x)]
[g(x) | x <- v]
```

are available as shortcuts for

```
apply(g, select(f, Vec(v)))
select(f, Vec(v))
apply(g, Vec(v))
```

respectively:

```
? L = List([Mod(1,3), Mod(2,4)]);
? [ lift(x) | x<-L ]
%2 = [1, 2]
```

arg(precision)

Argument of the complex number x, such that $-\pi < \arg(x) <= \pi$.

arity()

Return the arity of the closure C, i.e., the number of its arguments.

```
? f1(x,y=0)=x+y;
? arity(f1)
%1 = 2
? f2(t,s[..])=print(t,":",s);
? arity(f2)
%2 = 2
```

Note that a variadic argument, such as s in f2 above, is counted as a single argument.

asin(precision)

Principal branch of $\sin^{-1}(x) = -i \log(ix + \sqrt{1 - x^2})$. In particular, $\Re(asin(x)) \in [-\pi/2, \pi/2]$ and if $x \in \mathbb{R}$ and ||x|| > 1 then asin(x) is complex. The branch cut is in two pieces:]-oo, -1], continuous with quadrant II, and [1, +oo] continuous with quadrant IV. The function satisfies iasin(x) = asinh(ix).

asinh(precision)

Principal branch of $\sinh^{-1}(x) = \log(x + \sqrt{1 + x^2})$. In particular $\Im(asinh(x)) \in [-\pi/2, \pi/2]$. The

branch cut is in two pieces:] - ioo, -i], continuous with quadrant III and [+i, +ioo], continuous with quadrant I.

asympnum(alpha, precision)

Asymptotic expansion of *expr*, corresponding to a sequence u(n), assuming it has the shape

$$u(n) \sum_{i>=0} a_i n^{-i\alpha}$$

with rational coefficients a_i with reasonable height; the algorithm is heuristic and performs repeated calls to limitnum, with alpha as in limitnum. As in limitnum, u(n) may be given either by a closure n : -- u(n) or as a closure N : -- u(n), ..., u(N), the latter being often more efficient.

```
? f(n) = n! / (n^n*exp(-n)*sqrt(n));
? asympnum(f)
%2 = [] \\ failure !
? localprec(57); l = limitnum(f)
%3 = 2.5066282746310005024157652848110452530
? asympnum(n->f(n)/l) \\ normalize
%4 = [1, 1/12, 1/288, -139/51840, -571/2488320, 163879/209018880,
5246819/75246796800]
```

and we indeed get a few terms of Stirling's expansion. Note that it definitely helps to normalize with a limit computed to higher accuracy (as a rule of thumb, multiply the bit accuracy by 1.612):

```
? l = limitnum(f)
? asympnum(n->f(n) / l) \\ failure again !!!
%6 = []
```

We treat again the example of the Motzkin numbers M_n given in limitnum:

```
\\ [M_k, M_{k*2}, ..., M_{k*N}] / (3^n / n^(3/2))
? vM(N, k = 1) =
{ my(q = k*N, V);
    if (q == 1, return ([1/3]));
    V = vector(q); V[1] = V[2] = 1;
    for(n = 2, q - 1,
        V[n+1] = ((2*n + 1)*V[n] + 3*(n - 1)*V[n-1]) / (n + 2));
    f = (n -> 3^n / n^(3/2));
    return (vector(N, n, V[n*k] / f(n*k)));
}
? localprec(100); l = limitnum(n->vM(n,10)); \\ 3/sqrt(12*Pi)
? \p38
? asympnum(n->vM(n,10)/l)
%2 = [1, -3/32, 101/10240, -1617/1638400, 505659/5242880000, ...]
```

If alpha is not a rational number, loss of accuracy is expected, so it should be precomputed to double accuracy, say:

```
? \p38
? asympnum(n->log(1+1/n^Pi),Pi)
%1 = [0, 1, -1/2, 1/3, -1/4, 1/5]
? localprec(76); a = Pi;
? asympnum(n->log(1+1/n^Pi), a) \\ more terms
%3 = [0, 1, -1/2, 1/3, -1/4, 1/5, -1/6, 1/7, -1/8, 1/9, -1/10, 1/11, -1/12]
```

```
? asympnum(n->log(1+1/sqrt(n)),1/2) \setminus many more terms %4 = 49
```

The expression is evaluated for n = 1, 2, ..., N for an N = O(B) if the current bit accuracy is B. If it is not defined for one of these values, translate or rescale accordingly:

```
? asympnum(n->log(1-1/n)) \\ can't evaluate at n = 1 !
    *** at top-level: asympnum(n->log(1-1/n))
    *** ^------
    *** in function asympnum: log(1-1/n)
    *** ^------
    *** log: domain error in log: argument = 0
? asympnum(n->-log(1-1/(2*n)))
%5 = [0, 1/2, 1/8, 1/24, ...]
? asympnum(n->-log(1-1/(n+1)))
%6 = [0, 1, -1/2, 1/3, -1/4, ...]
```

asympnumraw(N, alpha, precision)

Return the N+1 first terms of asymptotic expansion of *expr*, corresponding to a sequence u(n), as floating point numbers. Assume that the expansion has the shape

$$u(n) \sum_{i>=0} a_i n^{-i\alpha}$$

and return approximation of $[a_0, a_1, ..., a_N]$. The algorithm is heuristic and performs repeated calls to limitnum, with alpha as in limitnum. As in limitnum, u(n) may be given either by a closure n : --- u(n) or as a closure N : -- v(n), the latter being often more efficient. This function is related to, but more flexible than, asymptour, which requires rational asymptotic expansions.

```
? f(n) = n! / (n^n*exp(-n)*sqrt(n));
? asympnum(f)
%2 = [] \\ failure !
? v = asympnumraw(f, 10);
? v[1] - sqrt(2*Pi)
%4 = 0.E-37
? bestappr(v / v[1], 2^60)
%5 = [1, 1/12, 1/288, -139/51840, -571/2488320, 163879/209018880,...]
```

and we indeed get a few terms of Stirling's expansion (the first 9 terms are correct). If u(n) has an asymptotic expansion in $n^{-\alpha}$ with α not an integer, the default alpha = 1 is inaccurate:

? f(n) = (1+1/n^(7/2))^(n^(7/2)); ? v1 = asympnumraw(f,10); ? v1[1] - exp(1) %8 = 4.62... E-12 ? v2 = asympnumraw(f,10,7/2); ? v2[1] - exp(1) %7 0.E-37

As in asymptoum, if alpha is not a rational number, loss of accuracy is expected, so it should be precomputed to double accuracy, say.

atan(precision)

Principal branch of $tan^{-1}(x) = \log((1+ix)/(1-ix))/2i$. In particular the real part of atan(x) belongs

to $] - \pi/2, \pi/2[$. The branch cut is in two pieces:] - ioo, -i[, continuous with quadrant IV, and]i, +ioo[continuous with quadrant II. The function satisfies atan(x) = -iatanh(ix) for all x! = i.

atanh(precision)

Principal branch of $tanh^{-1}(x) = \log((1+x)/(1-x))/2$. In particular the imaginary part of atanh(x) belongs to $[-\pi/2, \pi/2]$; if $x \in \mathbb{R}$ and ||x|| > 1 then atanh(x) is complex.

besselh1(*x*, *precision*)

 H^1 -Bessel function of index *nu* and argument *x*.

besselh2(x, precision)

 H^2 -Bessel function of index *nu* and argument *x*.

besseli(x, precision)

I-Bessel function of index *nu* and argument *x*. If *x* converts to a power series, the initial factor $(x/2)^{\nu}/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

besselj(x, precision)

J-Bessel function of index *nu* and argument *x*. If *x* converts to a power series, the initial factor $(x/2)^{\nu}/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

besseljh(x, precision)

J-Bessel function of half integral index. More precisely, bessel jh(n, x) computes $J_{n+1/2}(x)$ where n must be of type integer, and x is any element of \mathbb{C} . In the present version **2.13.3**, this function is not very accurate when x is small.

besselk(x, precision)

K-Bessel function of index nu and argument x.

besseln(x, precision)

Deprecated alias for bessely.

bessely(*x*, *precision*)

Y-Bessel function of index nu and argument x.

bestappr(B)

Using variants of the extended Euclidean algorithm, returns a rational approximation a/b to x, whose denominator is limited by B, if present. If B is omitted, returns the best approximation affordable given the input accuracy; if you are looking for true rational numbers, presumably approximated to sufficient accuracy, you should first try that option. Otherwise, B must be a positive real scalar (impose 0 < b <= B).

• If x is a t_REAL or a t_FRAC, this function uses continued fractions.

```
? bestappr(Pi, 100)
%1 = 22/7
? bestappr(0.1428571428571428571428571429)
%2 = 1/7
? bestappr([Pi, sqrt(2) + 'x], 10^3)
%3 = [355/113, x + 1393/985]
```

By definition, a/b is the best rational approximation to x if ||bx - a|| < ||vx - u|| for all integers (u, v) with $0 < v \le B$. (Which implies that n/d is a convergent of the continued fraction of x.)

If x is a t_INTMOD modulo N or a t_PADIC of precision N = p^k, this function performs rational modular reconstruction modulo N. The routine then returns the unique rational number a/b in coprime integers ||a|| < N/2B and b <= B which is congruent to x modulo N. Omitting B amounts to choosing it of the order of √N/2. If rational reconstruction is not possible (no suitable a/b exists), returns [].

```
? bestappr(Mod(18526731858, 11^10))
%1 = 1/7
? bestappr(Mod(18526731858, 11^20))
%2 = []
? bestappr(3 + 5 + 3*5^2 + 5^3 + 3*5^4 + 5^5 + 3*5^6 + 0(5^7))
%2 = -1/3
```

In most concrete uses, B is a prime power and we performed Hensel lifting to obtain x.

The function applies recursively to components of complex objects (polynomials, vectors,...). If rational reconstruction fails for even a single entry, returns [].

bestapprPade(B)

Using variants of the extended Euclidean algorithm (Padé approximants), returns a rational function approximation a/b to x, whose denominator is limited by B, if present. If B is omitted, return the best approximation affordable given the input accuracy; if you are looking for true rational functions, presumably approximated to sufficient accuracy, you should first try that option. Otherwise, B must be a nonnegative real (impose $0 \le degree(b) \le B$).

• If x is a t_POLMOD modulo N this function performs rational modular reconstruction modulo N. The routine then returns the unique rational function a/b in coprime polynomials, with $degree(b) \le B$ and degree(a) minimal, which is congruent to x modulo N. Omitting B amounts to choosing it equal to the floor of degree(N)/2. If rational reconstruction is not possible (no suitable a/b exists), returns [].

```
? T = Mod(x<sup>3</sup> + x<sup>2</sup> + x + 3, x<sup>4</sup> - 2);
? bestapprPade(T)
%2 = (2*x - 1)/(x - 1)
? U = Mod(1 + x + x<sup>2</sup> + x<sup>3</sup> + x<sup>5</sup>, x<sup>9</sup>);
? bestapprPade(U) \\ internally chooses B = 4
%3 = []
? bestapprPade(U, 5) \\ with B = 5, a solution exists
%4 = (2*x<sup>4</sup> + x<sup>3</sup> - x - 1)/(-x<sup>5</sup> + x<sup>3</sup> + x<sup>2</sup> - 1)
```

• If x is a t_SER, we implicitly convert the input to a t_POLMOD modulo $N = t^k$ where k is the series absolute precision.

? $T = 1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + 0(t^7); \ \mod t^7$? bestapprPade(T) %1 = 1/(-t + 1)

• If x is a t_RFRAC, we implicitly convert the input to a t_POLMOD modulo $N = t^k$ where k = 2B + 1. If B was omitted, we return x:

```
? T = (4*t<sup>2</sup> + 2*t + 3)/(t+1)<sup>10</sup>;
? bestapprPade(T,1)
%2 = [] \\ impossible
? bestapprPade(T,2)
%3 = 27/(337*t<sup>2</sup> + 84*t + 9)
? bestapprPade(T,3)
%4 = (4253*t - 3345)/(-39007*t<sup>3</sup> - 28519*t<sup>2</sup> - 8989*t - 1115)
```

The function applies recursively to components of complex objects (polynomials, vectors,...). If rational reconstruction fails for even a single entry, return [].

bestapprnf(T, rootT, precision)

T being an integral polynomial and *V* being a scalar, vector, or matrix with complex coefficients, return a reasonable approximation of *V* with polmods modulo *T*. *T* can also be any number field structure, in which case the minimal polynomial attached to the structure (:math:`T`.pol) is used. The *rootT* argument, if present, must be an element of polroots(:math:`T)` (or :math:`T`.pol), i.e. a complex root of *T* fixing an embedding of $\mathbb{Q}[x]/(T)$ into \mathbb{C} .

```
? bestapprnf(sqrt(5), polcyclo(5))
%1 = Mod(-2*x^3 - 2*x^2 - 1, x^4 + x^3 + x^2 + x + 1)
? bestapprnf(sqrt(5), polcyclo(5), exp(4*I*Pi/5))
%2 = Mod(2*x^3 + 2*x^2 + 1, x^4 + x^3 + x^2 + x + 1)
```

When the output has huge rational coefficients, try to increase the working realbitprecision: if the answer does not stabilize, consider that the reconstruction failed. Beware that if T is not Galois over \mathbb{Q} , some embeddings may not allow to reconstruct V:

```
? T = x^3-2; vT = polroots(T); z = 3*2^(1/3)+1;
? bestapprnf(z, T, vT[1])
%2 = Mod(3*x + 1, x^3 - 2)
? bestapprnf(z, T, vT[2])
%3 = 4213714286230872/186454048314072 \\ close to 3*2^(1/3) + 1
```

bezout(y)

Deprecated alias for gcdext

bezoutres(B, v)

Deprecated alias for polresultantext

bigomega()

Number of prime divisors of the integer ||x|| counted with multiplicity:

```
? factor(392)
%1 =
[2 3]
[7 2]
? bigomega(392)
%2 = 5; \\ = 3+2
? omega(392)
%3 = 2; \\ without multiplicity
```

binary()

Outputs the vector of the binary digits of ||x||. Here x can be an integer, a real number (in which case the result has two components, one for the integer part, one for the fractional part) or a vector/matrix.

```
? binary(10)
%1 = [1, 0, 1, 0]
? binary(3.14)
%2 = [[1, 1], [0, 0, 1, 0, 0, 0, [...]]
```

? binary([1,2]) %3 = [[1], [1, 0]]

For integer $x \ge 1$, the number of bits is logint(x, 2) + 1. By convention, 0 has no digits:

? binary(0) %4 = []

binomial(k)

binomial coefficient binomxk. Here k must be an integer, but x can be any PARI object.

? binomial(4,2)
%1 = 6
? n = 4; vector(n+1, k, binomial(n,k-1))
%2 = [1, 4, 6, 4, 1]

The argument k may be omitted if x = n is a nonnegative integer; in this case, return the vector with n + 1 components whose k + 1-th entry is binomial(n, k)

? binomial(4) %3 = [1, 4, 6, 4, 1]

bitand(y)

Bitwise and of two integers x and y, that is the integer

$$\sum_{i} (x_i \text{ and } y_i) 2^i$$

Negative numbers behave 2-adically, i.e. the result is the 2-adic limit of $bitand(x_n, y_n)$, where x_n and y_n are nonnegative integers tending to x and y respectively. (The result is an ordinary integer, possibly negative.)

? bitand(5, 3)
%1 = 1
? bitand(-5, 3)
%2 = 3
? bitand(-5, -3)
%3 = -7

bitneg(n)

bitwise negation of an integer x, truncated to n bits, $n \ge 0$, that is the integer

$$\sum_{i=0}^{n-1} not(x_i) 2^i.$$

The special case n = -1 means no truncation: an infinite sequence of leading 1 is then represented as a negative number.

See bit and (in the PARI manual) for the behavior for negative arguments.

bitnegimply(y)

Bitwise negated imply of two integers x and y (or not (x = y)), that is the integer

$$\sum (x_i \ and not(y_i)) 2^i$$

See bit and (in the PARI manual) for the behavior for negative arguments.

bitor(y)

bitwise (inclusive) or of two integers x and y, that is the integer

$$\sum (x_i \text{ or } y_i) 2^i$$

See bitand (in the PARI manual) for the behavior for negative arguments.

bitprecision(n)

The function behaves differently according to whether n is present or not. If n is missing, the function returns the (floating point) precision in bits of the PARI object x.

If x is an exact object, the function returns +00.

```
? bitprecision(exp(1e-100))
%1 = 512 \\ 512 bits
? bitprecision( [ exp(1e-100), 0.5 ] )
%2 = 128 \\ minimal accuracy among components
? bitprecision(2 + x)
%3 = +00 \\ exact object
```

Use getlocalbitprec() to retrieve the working bit precision (as modified by possible localbitprec statements).

If n is present and positive, the function creates a new object equal to x with the new bit-precision roughly n. In fact, the smallest multiple of 64 (resp. 32 on a 32-bit machine) larger than or equal to n.

For x a vector or a matrix, the operation is done componentwise; for series and polynomials, the operation is done coefficientwise. For real x, n is the number of desired significant *bits*. If n is smaller than the precision of x, x is truncated, otherwise x is extended with zeros. For exact or non-floating-point types, no change.

```
? bitprecision(Pi, 10) \\ actually 64 bits ~ 19 decimal digits
%1 = 3.141592653589793239
? bitprecision(1, 10)
%2 = 1
? bitprecision(1 + 0(x), 10)
%3 = 1 + 0(x)
? bitprecision(2 + 0(3^5), 10)
%4 = 2 + 0(3^5)
```

bittest(n)

Outputs the n - th bit of x starting from the right (i.e. the coefficient of 2^n in the binary expansion of x). The result is 0 or 1. For $x \ge 1$, the highest 1-bit is at n = logint(x) (and bigger n gives 0).

```
? bittest(7, 0)
%1 = 1 \\ the bit 0 is 1
? bittest(7, 2)
%2 = 1 \\ the bit 2 is 1
? bittest(7, 3)
%3 = 0 \\ the bit 3 is 0
```

See bitand (in the PARI manual) for the behavior at negative arguments.

bitxor(y)

Bitwise (exclusive) or of two integers x and y, that is the integer

$$\sum (x_i \ xor \ y_i) 2^i$$

See bitand (in the PARI manual) for the behavior for negative arguments.

bnfcertify(flag)

bnf being as output by bnfinit, checks whether the result is correct, i.e. whether it is possible to remove the assumption of the Generalized Riemann Hypothesis. It is correct if and only if the answer is 1. If it is incorrect, the program may output some error message, or loop indefinitely. You can check its progress by increasing the debug level. The *bnf* structure must contain the fundamental units:

```
? K = bnfinit(x^3+2^2^3+1); bnfcertify(K)
 *** at top-level: K=bnfinit(x^3+2^2^3+1); bnfcertify(K)
 *** ^------
 *** bnfcertify: precision too low in makeunits [use bnfinit(,1)].
? K = bnfinit(x^3+2^2^3+1, 1); \\ include units
? bnfcertify(K)
%3 = 1
```

If flag is present, only certify that the class group is a quotient of the one computed in bnf (much simpler in general); likewise, the computed units may form a subgroup of the full unit group. In this variant, the units are no longer needed:

```
? K = bnfinit(x^3+2^2*3+1); bnfcertify(K, 1)
%4 = 1
```

bnfdecodemodule(m)

If m is a module as output in the first component of an extension given by bnrdisclist, outputs the true module.

```
? K = bnfinit(x^2+23); L = bnrdisclist(K, 10); s = L[2]
%1 = [[[Vecsmall([8]), Vecsmall([1])], [[0, 0, 0]]],
  [[Vecsmall([9]), Vecsmall([1])], [[0, 0, 0]]]]
? bnfdecodemodule(K, s[1][1])
%2 =
[2 0]
[0 1]
? bnfdecodemodule(K,s[2][1])
%3 =
[2 1]
[0 1]
```

bnfinit(flag, tech, precision)

Initializes a bnf structure. Used in programs such as bnfisprincipal, bnfisunit or bnfnarrow. By default, the results are conditional on the GRH, see GRHbnf (in the PARI manual). The result is a 10-component vector *bnf*.

This implements Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible polynomial P with integer coefficients. The meaning of *flag* is as follows:

• *flag* = 0 (default). This is the historical behavior, kept for compatibility reasons and speed. It has severe drawbacks but is likely to be a little faster than the alternative, twice faster say, so only use it if speed is paramount, you obtain a useful speed gain for the fields under consideration, and you are only interested in the field invariants such as the classgroup structure or its regulator. The computations involve exact algebraic numbers which are replaced by floating point embeddings for the sake of speed. If the precision is insufficient, gp may not be able to compute fundamental units, nor to solve some discrete logarithm problems. It *may* be possible to increase the precision of the bnf structure using nfnewprec but this may fail, in particular when fundamental units are large. In short, the resulting bnf structure is correct and contains useful information but later function calls to bnfisprincpal or bnrclassfield may fail.

When flag = 1, we keep an exact algebraic version of all floating point data and this allows to guarantee that functions using the structure will always succeed, as well as to compute the fundamental units exactly. The units are computed in compact form, as a product of small S-units, possibly with huge exponents. This flag also allows bnfisprincipal to compute generators of principal ideals in factored form as well. Be warned that expanding such products explicitly can take a very long time, but they can easily be mapped to floating point or ℓ -adic embeddings of bounded accuracy, or to $K^*/(K^*)^{\ell}$, and this is enough for applications. In short, this flag should be used by default, unless you have a very good reason for it, for instance building massive tables of class numbers, and you do not care about units or the effect large units would have on your computation.

tech is a technical vector (empty by default, see GRHbnf (in the PARI manual)). Careful use of this parameter may speed up your computations, but it is mostly obsolete and you should leave it alone.

The components of a *bnf* are technical. In fact: *never access a component directly, always use a proper member function.* However, for the sake of completeness and internal documentation, their description is as follows. We use the notations explained in the book by H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Maths **138**, Springer-Verlag, 1993, Section 6.5, and subsection 6.5.5 in particular.

bnf[1] contains the matrix W, i.e. the matrix in Hermite normal form giving relations for the class group on prime ideal generators $(p_i)_{1 \le i \le r}$.

bnf[2] contains the matrix B, i.e. the matrix containing the expressions of the prime ideal factorbase in terms of the p_i . It is an rxc matrix.

bnf[3] contains the complex logarithmic embeddings of the system of fundamental units which has been found. It is an $(r_1 + r_2)x(r_1 + r_2 - 1)$ matrix.

bnf[4] contains the matrix M''_C of Archimedean components of the relations of the matrix (W||B).

bnf[5] contains the prime factor base, i.e. the list of prime ideals used in finding the relations.

bnf[6] contains a dummy 0.

bnf[7] or :emphasis: `bnf.nf` is equal to the number field data nf as would be given by nfinit.

bnf[8] is a vector containing the classgroup :emphasis: `bnf.clgp` as a finite abelian group, the regulator :emphasis: `bnf.reg`, the number of roots of unity and a generator :emphasis: `bnf.tu`, the fundamental units *in expanded form* :emphasis: `bnf.fu`. If the fundamental units were omitted in the *bnf*, :emphasis: `bnf.fu` returns the sentinel value 0. If flag = 1, this vector contain also algebraic data corresponding to the fundamental units and to the discrete logarithm problem (see bnfisprincipal). In particular, if flag = 1 we may *only* know the units in factored form: the first call to :emphasis: `bnf.fu` expands them, which may be very costly, then caches the result.

bnf[9] is a vector used in bnfisprincipal only and obtained as follows. Let D = UWV obtained by applying the Smith normal form algorithm to the matrix W (= bnf[1]) and let U_r be the reduction of U modulo D. The first elements of the factorbase are given (in terms of bnf.gen) by the columns of U_r , with Archimedean component g_a ; let also GD_a be the Archimedean components of the generators of the
(principal) ideals defined by the bnf.gen[i]^bnf.cyc[i]. Then $bnf[9] = [U_r, g_a, GD_a]$, followed by technical exact components which allow to recompute g_a and GD_a to higher accuracy.

bnf[10] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available, which is rarely needed, hence would be too expensive to compute during the initial bnfinit call. For instance, the generators of the principal ideals bnf.gen[i]^bnf.cyc[i] (during a call to bnrisprincipal), or those corresponding to the relations in W and B (when the bnf internal precision needs to be increased).

bnfisintnorm(x)

Computes a complete system of solutions (modulo units of positive norm) of the absolute norm equation Norm(a) = x, where a is an integer in bnf. If bnf has not been certified, the correctness of the result depends on the validity of GRH.

See also bnfisnorm.

bnfisnorm(x, flag)

Tries to tell whether the rational number x is the norm of some element y in bnf. Returns a vector [a, b] where x = Norm(a) * b. Looks for a solution which is an S-unit, with S a certain set of prime ideals containing (among others) all primes dividing x. If bnf is known to be Galois, you may set flag = 0 (in this case, x is a norm iff b = 1). If flag is nonzero the program adds to S the following prime ideals, depending on the sign of flag. If flag > 0, the ideals of norm less than flag. And if flag < 0 the ideals dividing flag.

Assuming GRH, the answer is guaranteed (i.e. x is a norm iff b = 1), if S contains all primes less than $12 \log(\operatorname{disc}(Bnf))^2$, where Bnf is the Galois closure of bnf.

See also bnfisintnorm.

bnfisprincipal(x, flag)

bnf being the number field data output by bnfinit, and x being an ideal, this function tests whether the ideal is principal or not. The result is more complete than a simple true/false answer and solves a general discrete logarithm problem. Assume the class group is $\oplus (\mathbb{Z}/d_i\mathbb{Z})g_i$ (where the generators g_i and their orders d_i are respectively given by bnf.gen and bnf.cyc). The routine returns a row vector [e, t], where e is a vector of exponents $0 \le e_i < d_i$, and t is a number field element such that

$$x = (t) \prod_{i} g_i^{e_i}.$$

For given g_i (i.e. for a given bnf), the e_i are unique, and t is unique modulo units.

In particular, x is principal if and only if e is the zero vector. Note that the empty vector, which is returned when the class number is 1, is considered to be a zero vector (of dimension 0).

```
? K = bnfinit(y^2+23);
? K.cyc
%2 = [3]
? K.gen
%3 = [[2, 0; 0, 1]] \\ a prime ideal above 2
? P = idealprimedec(K,3)[1]; \\ a prime ideal above 3
? v = bnfisprincipal(K, P)
%5 = [[2]~, [3/4, 1/4]~]
? idealmul(K, v[2], idealfactorback(K, K.gen, v[1]))
%6 =
[3 0]
[0 1]
```

? % == idealhnf(K, P) %7 = 1

The binary digits of *flag* mean:

• 1: If set, outputs [e, t] as explained above, otherwise returns only e, which is much easier to compute. The following idiom only tests whether an ideal is principal:

```
is_principal(bnf, x) = !bnfisprincipal(bnf,x,0);
```

- 2: It may not be possible to recover t, given the initial accuracy to which the bnf structure was computed. In that case, a warning is printed and t is set equal to the empty vector []~. If this bit is set, increase the precision and recompute needed quantities until t can be computed. Warning: setting this may induce *lengthy* computations and you should consider using flag 4 instead.
- 4: Return t in factored form (compact representation), as a small product of S-units for a small set of finite places S, possibly with huge exponents. This kind of result can be cheaply mapped to K*/(K*)^ℓ or to C or Q_p to bounded accuracy and this is usually enough for applications. Explicitly expanding such a compact representation is possible using nffactorback but may be very costly. The algorithm is guaranteed to succeed if the bnf was computed using bnfinit(,1). If not, the algorithm may fail to compute a huge generator in this case (and replace it by []~). This is orders of magnitude faster than flag 2 when the generators are indeed large.

bnfissunit(sfu, x)

This function is obsolete, use bnfisunit.

bnfisunit(x, U)

but being the number field data output by bnfinit and x being an algebraic number (type integer, rational or polmod), this outputs the decomposition of x on the fundamental units and the roots of unity if x is a unit, the empty vector otherwise. More precisely, if $u_1, \ldots, :math: u_r$ are the fundamental units, and ζ is the generator of the group of roots of unity (bnf.tu), the output is a vector $[x_1, \ldots, x_r, x_{r+1}]$ such that $x = u_1^{x_1} \ldots u_r^{x_r} . \zeta^{x_{r+1}}$. The x_i are integers but the last one (i = r + 1) is only defined modulo the order w of ζ and is guaranteed to be in [0, w].

Note that *bnf* need not contain the fundamental units explicitly: it may contain the placeholder 0 instead:

```
? setrand(1); bnf = bnfinit(x^2-x-100000);
? bnf.fu
%2 = 0
? u = [119836165644250789990462835950022871665178127611316131167, \
379554884019013781006303254896369154068336082609238336]~;
? bnfisunit(bnf, u)
%3 = [-1, 0]~
```

The given u is $1/u_1$, where u_1 is the fundamental unit implicitly stored in *bnf*. In this case, u_1 was not computed and stored in algebraic form since the default accuracy was too low. Re-run the bnfinit command at \g1 or higher to see such diagnostics.

This function allows x to be given in factored form, but it then assumes that x is an actual unit. (Because it is general too costly to check whether this is the case.)

```
? { v = [2, 85; 5, -71; 13, -162; 17, -76; 23, -37; 29, -104; [224, 1] \sim, -66; [-86, 1] \sim, 86; [-241, 1] \sim, -20; [44, 1] \sim, 30; [124, 1] \sim, 11; [125, -1] \sim, -11; [-214, 1] \sim, 33; [-213, -1] \sim, -33; [189, 1] \sim, 74; [190, -1] \sim, 104;
```

```
[-168, 1]~, 2; [-167, -1]~, -8]; }
? bnfisunit(bnf,v)
%5 = [1, 0]~
```

Note that v is the fundamental unit of **bnf** given in compact (factored) form.

If the argument U is present, as output by bnfunits(bnf, S), then the function decomposes x on the S-units generators given in U[1].

```
? bnf = bnfinit(x^4 - x^3 + 4*x^2 + 3*x + 9, 1);
? bnf.sign
\%2 = [0, 2]
? S = idealprimedec(bnf,5); #S
%3 = 2
? US = bnfunits(bnf,S);
? g = US[1]; #g \\ #S = #g, four S-units generators, in factored form
%5 = 4
? g[1]
\%6 = [[6, -3, -2, -2] \sim 1]
? g[2]
%7 =
[[-1, 1/2, -1/2, -1/2]~ 1]
[ [4, -2, -1, -1] \sim 1 ]
? [nffactorback(bnf, x) | x <- g]</pre>
\%8 = [[6, -3, -2, -2] \sim, [-5, 5, 0, 0] \sim, [-1, 1, -1, 0] \sim,
[1, -1, 0, 0] \sim ]
? u = [10, -40, 24, 11] \sim;
? a = bnfisunit(bnf, u, US)
\%9 = [2, 0, 1, 4] \sim
? nffactorback(bnf, g, a) \\ prod_i g[i]^a[i] still in factored form
%10 =
[[6, -3, -2, -2]~ 2]
[ [0, 0, -1, -1]~ 1]
[[2, -1, -1, 0] \sim -2]
[ [1, 1, 0, 0]~ 2]
[ [-1, 1, 1, 1]~ -1]
[ [1, -1, 0, 0] \sim 4 ]
? nffactorback(bnf,%) \\ u = prod_i g[i]^a[i]
%11 = [10, -40, 24, 11]~
```

bnflog(*l*)

Let *bnf* be a *bnf* structure attached to the number field F and let l be a prime number (hereafter denoted ℓ for typographical reasons). Return the logarithmic ℓ -class group Cl_F of F. This is an abelian group, conjecturally finite (known to be finite if F/\mathbb{Q} is abelian). The function returns if and only if the group is indeed finite (otherwise it would run into an infinite loop). Let $S = p_1, ..., p_k$ be the set of ℓ -adic places

(maximal ideals containing ℓ). The function returns $[D, G(\ell), G']$, where

- D is the vector of elementary divisors for Cl_F .
- $G(\ell)$ is the vector of elementary divisors for the (conjecturally finite) abelian group

where the :math: p_i 'are the : math : ℓ' -adicplaces of : math : 'F'; this is a subgroup of : math : 'Cl'.

• G' is the vector of elementary divisors for the ℓ -Sylow Cl' of the S-class group of F; the group Cl maps to Cl' with a simple co-kernel.

bnflogdegree(A, l)

Let *nf* be a *nf* structure attached to a number field *F*, and let *l* be a prime number (hereafter denoted ℓ). The ℓ -adified group of id\`{e}les of *F* quotiented by the group of logarithmic units is identified to the ℓ -group of logarithmic divisors $\oplus \mathbb{Z}_{\ell}[p]$, generated by the maximal ideals of *F*.

The *degree* map deg_F is additive with values in \mathbb{Z}_{ℓ} , defined by deg_F $p = f_p \deg_{\ell} p$, where the integer f_p is as in **bnflogef** and deg_{ℓ} p is $\log_{\ell} p$ for $p! = \ell$, $\log_{\ell}(1 + \ell)$ for $p = \ell! = 2$ and $\log_{\ell}(1 + 2^2)$ for $p = \ell = 2$.

Let $A = \prod p^{n_p}$ be an ideal and let $A = \sum n_p[p]$ be the attached logarithmic divisor. Return the exponential of the ℓ -adic logarithmic degree deg_F A, which is a natural number.

bnflogef(pr)

Let *nf* be a *nf* structure attached to a number field *F* and let *pr* be a *prid* structure attached to a maximal ideal p/p. Return $[e(F_p/\mathbb{Q}_p), f(F_p/\mathbb{Q}_p)]$ the logarithmic ramification and residue degrees. Let $\mathbb{Q}_p^c/\mathbb{Q}_p$ be the cyclotomic \mathbb{Z}_p -extension, then $e = [F_p : F_p \cap \mathbb{Q}_p^c]$ and $f = [F_p \cap \mathbb{Q}_p^c : \mathbb{Q}_p]$. Note that e f = e(p/p)f(p/p), where e(p/p) and f(p/p) denote the usual ramification and residue degrees.

```
? F = nfinit(y^6 - 3*y^5 + 5*y^3 - 3*y + 1);
? bnflogef(F, idealprimedec(F,2)[1])
%2 = [6, 1]
? bnflogef(F, idealprimedec(F,5)[1])
%3 = [1, 2]
```

bnfnarrow()

bnf being as output by bnfinit, computes the narrow class group of *bnf*. The output is a 3-component row vector v analogous to the corresponding class group component :emphasis: `bnf.clgp`: the first component is the narrow class number :math: `v.no`, the second component is a vector containing the SNF cyclic components :math: `v.cyc` of the narrow class group, and the third is a vector giving the generators of the corresponding :math: `v.gen` cyclic groups. Note that this function is a special case of bnrinit; the *bnf* need not contain fundamental units.

bnfsignunit()

bnf being as output by bnfinit, this computes an $r_1x(r_1 + r_2 - 1)$ matrix having 1 components, giving the signs of the real embeddings of the fundamental units. The following functions compute generators for the totally positive units:

```
/* exponents of totally positive units generators on K.tu, K.fu */
tpuexpo(K)=
{ my(M, S = bnfsignunit(K), [m,n] = matsize(S));
    \\ m = K.r1, n = r1+r2-1
    S = matrix(m,n, i,j, if (S[i,j] < 0, 1,0));</pre>
```

```
S = concat(vectorv(m,i,1), S); \\ add sign(-1)
M = matkermod(S, 2);
if (M, mathnfmodid(M, 2), 2*matid(n+1))
}
/* totally positive fundamental units of bnf K */
tpu(K)=
{ my(ex = tpuexpo(K)[,^1]); \\ remove ex[,1], corresponds to 1 or -1
my(v = concat(K.tu[2], K.fu));
[ nffactorback(K, v, c) | c <- ex];
}</pre>
```

bnfsunit(S, precision)

Computes the fundamental S-units of the number field bnf (output by bnfinit), where S is a list of prime ideals (output by idealprimedec). The output is a vector v with 6 components.

v[1] gives a minimal system of (integral) generators of the S-unit group modulo the unit group.

v[2] contains technical data needed by bnfissunit.

v[3] is an obsoleted component, now the empty vector.

v[4] is the S-regulator (this is the product of the regulator, the S-class number and the natural logarithms of the norms of the ideals in S).

v[5] gives the S-class group structure, in the usual abelian group format: a vector whose three components give in order the S-class number, the cyclic components and the generators.

v[6] is a copy of S.

bnfunits(S)

Return the fundamental units of the number field bnf output by bnfinit; if S is present and is a list of prime ideals, compute fundamental S-units instead. The first component of the result contains independent integral S-units generators: first nonunits, then $r_1 + r_2 - 1$ fundamental units, then the torsion unit. The result may be used as an optional argument to bnfisunit. The units are given in compact form: no expensive computation is attempted if the *bnf* does not already contain units.

```
? bnf = bnfinit(x^4 - x^3 + 4*x^2 + 3*x + 9, 1);
? bnf.sign \\ r1 + r2 - 1 = 1
%2 = [0, 2]
? U = bnfunits(bnf); u = U[1];
? #u \\ r1 + r2 = 2 units
%5 = 2;
? u[1] \\ fundamental unit as factorization matrix
%6 =
[[0, 0, -1, -1]~ 1]
[[2, -1, -1, 0]~ -2]
[ [1, 1, 0, 0]~ 2]
[ [-1, 1, 1, 1]~ -1]
? u[2] \\ torsion unit as factorization matrix
%7 =
[[1, -1, 0, 0]~ 1]
```

? [nffactorback(bnf, z) | z <- u] $\$ same units in expanded form %8 = [[-1, 1, -1, 0]~, [1, -1, 0, 0]~]

Now an example involving S-units for a nontrivial S:

? S = idealprimedec(bnf,5); #S %9 = 2 ? US = bnfunits(bnf, S); uS = US[1]; ? g = [nffactorback(bnf, z) | z <- uS] \\ now 4 units</pre> 11 = [[6, -3, -2, -2] -, [-5, 5, 0, 0] -, [-1, 1, -1, 0] -, [1, -1, 0, 0] -]? bnfisunit(bnf,[10,-40,24,11]~) $\%12 = [] \sim \setminus$ not a unit ? e = bnfisunit(bnf, [10,-40,24,11]~, US) %13 = [2, 0, 1, 4]~ \\ ...but an S-unit ? nffactorback(bnf, g, e) %14 = [10, -40, 24, 11]~ ? nffactorback(bnf, uS, e) \\ in factored form %15 = [[6, -3, -2, -2]~ 2] $[[0, 0, -1, -1] \sim 1]$ $[[2, -1, -1, 0] \sim -2]$ [[1, 1, 0, 0]~ 2] [[-1, 1, 1, 1]~ -1] [[1, -1, 0, 0]~ 4]

Note that in more complicated cases, any nffactorback fully expanding an element in factored form could be *very* expensive. On the other hand, the final example expands a factorization whose components are themselves in factored form, hence the result is a factored form: this is a cheap operation.

bnrL1(H, flag, precision)

Let *bnr* be the number field data output by **bnrinit** and *H* be a square matrix defining a congruence subgroup of the ray class group corresponding to *bnr* (the trivial congruence subgroup if omitted). This function returns, for each character χ of the ray class group which is trivial on *H*, the value at s = 1 (or s = 0) of the abelian *L*-function attached to χ . For the value at s = 0, the function returns in fact for each χ a vector $[r_{\chi}, c_{\chi}]$ where

$$L(s,\chi) = c.s^r + O(s^{r+1})$$

near 0.

The argument *flag* is optional, its binary digits mean 1: compute at s = 0 if unset or s = 1 if set, 2: compute the primitive *L*-function attached to χ if unset or the *L*-function with Euler factors at prime ideals dividing the modulus of *bnr* removed if set (that is $L_S(s, \chi)$, where *S* is the set of infinite places of the number field together with the finite prime ideals dividing the modulus of *bnr*), 3: return also the character if set.

```
K = bnfinit(x^2-229);
bnr = bnrinit(K,1);
bnrL1(bnr)
```

returns the order and the first nonzero term of $L(s, \chi)$ at s = 0 where χ runs through the characters of the class group of $K = \mathbb{Q}(\sqrt{229})$. Then

bnr2 = bnrinit(K,2); bnrL1(bnr2,,2)

returns the order and the first nonzero terms of $L_S(s, \chi)$ at s = 0 where χ runs through the characters of the class group of K and S is the set of infinite places of K together with the finite prime 2. Note that the ray class group modulo 2 is in fact the class group, so bnrL1(bnr2,0) returns the same answer as bnrL1(bnr,0).

This function will fail with the message

*** bnrL1: overflow in zeta_get_N0 [need too many primes].

if the approximate functional equation requires us to sum too many terms (if the discriminant of K is too large).

bnrchar(g, v)

Returns all characters χ on bnr.clgp such that $\chi(g_i) = e(v_i)$, where $e(x) = \exp(2i\pi x)$. If v is omitted, returns all characters that are trivial on the g_i . Else the vectors g and v must have the same length, the g_i must be ideals in any form, and each v_i is a rational number whose denominator must divide the order of g_i in the ray class group. For convenience, the vector of the g_i can be replaced by a matrix whose columns give their discrete logarithm, as given by bnrisprincipal; this allows to specify abstractly a subgroup of the ray class group.

? bnr = bnrinit(bnfinit(x), [160,[1]], 1); /* (Z/160Z)^* */ ? bnr.cyc %2 = [8, 4, 2] ? g = bnr.gen; ? bnrchar(bnr, g, [1/2,0,0]) %4 = [[4, 0, 0]] \\ a unique character ? bnrchar(bnr, [g[1],g[3]]) \\ all characters trivial on g[1] and g[3] %5 = [[0, 1, 0], [0, 2, 0], [0, 3, 0], [0, 0, 0]] ? bnrchar(bnr, [1,0,0;0,1,0;0,0,2]) %6 = [[0, 0, 1], [0, 0, 0]] \\ characters trivial on given subgroup

bnrclassfield(subgp, flag, precision)

bnr being as output by bnrinit, returns a relative equation for the class field corresponding to the congruence group defined by (bnr, subgp) (the full ray class field if *subgp* is omitted). The subgroup can also be a t_INT *n*, meaning $n.Cl_f$. The function also handles a vector of subgroup, e.g, from subgrouplist and returns the vector of individual results in this case.

If flag = 0, returns a vector of polynomials such that the compositum of the corresponding fields is the class field; if flag = 1 returns a single polynomial; if flag = 2 returns a single absolute polynomial.

```
? bnf = bnfinit(y<sup>3</sup>+14*y-1); bnf.cyc
%1 = [4, 2]
? pol = bnrclassfield(bnf, 1) \\ Hilbert class field
%2 = x<sup>8</sup> - 2*x<sup>7</sup> + ... + Mod(11*y<sup>2</sup> - 82*y + 116, y<sup>3</sup> + 14*y - 1)
? rnfdisc(bnf,pol)[1]
%3 = 1
? bnr = bnrinit(bnf,3*5*7); bnr.cyc
%4 = [24, 12, 12, 2]
? bnrclassfield(bnr,2) \\ maximal 2-elementary subextension
%5 = [x<sup>2</sup> + (-21*y - 105), x<sup>2</sup> + (-5*y - 25), x<sup>2</sup> + (-y - 5), x<sup>2</sup> + (-y - 1)]
\\ quadratic extensions of maximal conductor
? bnrclassfield(bnr, subgrouplist(bnr,[2]))
%6 = [[x<sup>2</sup> - 105], [x<sup>2</sup> + (-105*y<sup>2</sup> - 1260)], [x<sup>2</sup> + (-105*y - 525)],
[x<sup>2</sup> + (-105*y - 105)]]
```

```
? #bnrclassfield(bnr,subgrouplist(bnr,[2],1)) \\ all quadratic extensions
%7 = 15
```

When the subgroup contains nCl_f , where n is fixed, it is advised to directly compute the bnr modulo n to avoid expensive discrete logarithms:

```
? bnf = bnfinit(y^2-5); p = 1594287814679644276013;
? bnr = bnrinit(bnf,p); \\ very slow
time = 24,146 ms.
? bnrclassfield(bnr, 2) \\ ... even though the result is trivial
%3 = [x^2 - 1594287814679644276013]
? bnr2 = bnrinit(bnf,p,,2); \\ now fast
time = 1 ms.
? bnrclassfield(bnr2, 2)
%5 = [x^2 - 1594287814679644276013]
```

This will save a lot of time when the modulus contains a maximal ideal whose residue field is large.

bnrclassno(B, C)

Let A, B, C define a class field L over a ground field K (of type [:emphasis:`bnr]`, [:emphasis:`bnr, subgroup]`, or [:emphasis:`bnf, modulus]`, or [:emphasis:`bnf, modulus,:emphasis:subgroup]`, CFT (in the PARI manual)); this function returns the relative degree [L:K].

In particular if A is a *bnf* (with units), and B a modulus, this function returns the corresponding ray class number modulo B. One can input the attached *bid* (with generators if the subgroup C is non trivial) for B instead of the module itself, saving some time.

This function is faster than bnrinit and should be used if only the ray class number is desired. See bnrclassnolist if you need ray class numbers for all moduli less than some bound.

bnrclassnolist(list)

bnf being as output by **bnfinit**, and *list* being a list of moduli (with units) as output by **ideallist** or **ideallistarch**, outputs the list of the class numbers of the corresponding ray class groups. To compute a single class number, **bnrclassno** is more efficient.

```
? bnf = bnfinit(x^2 - 2);
? L = ideallist(bnf, 100, 2);
? H = bnrclassnolist(bnf, L);
? H[98]
%4 = [1, 3, 1]
? l = L[1][98]; ids = vector(#l, i, l[i].mod[1])
%5 = [[98, 88; 0, 1], [14, 0; 0, 7], [98, 10; 0, 1]]
```

The weird l[i].mod[1], is the first component of l[i].mod, i.e. the finite part of the conductor. (This is cosmetic: since by construction the Archimedean part is trivial, I do not want to see it). This tells us that the ray class groups modulo the ideals of norm 98 (printed as %5) have respectively order 1, 3 and 1. Indeed, we may check directly:

```
? bnrclassno(bnf, ids[2])
%6 = 3
```

bnrconductor(*B*, *C*, *flag*)

Conductor f of the subfield of a ray class field as defined by [A, B, C] (of type [:emphasis:`bnr]`, [:emphasis:`bnr, subgroup]`, [:emphasis:`bnf, modulus]` or [:emphasis:`bnf, modulus, subgroup]`, CFT (in the PARI manual))

If flag = 0, returns f.

If flag = 1, returns $[f, Cl_f, H]$, where Cl_f is the ray class group modulo f, as a finite abelian group; finally H is the subgroup of Cl_f defining the extension.

If flag = 2, returns [f, bnr(f), H], as above except Cl_f is replaced by a bnr structure, as output by bnrinit(, f), without generators unless the input contained a *bnr* with generators.

In place of a subgroup H, this function also accepts a character $chi = (a_j)$, expressed as usual in terms of the generators bnr.gen: $\chi(g_j) = \exp(2i\pi a_j/d_j)$, where g_j has order $d_j = bnr.cyc[j]$. In which case, the function returns respectively

If flag = 0, the conductor f of $Ker\chi$.

If flag = 1, $[f, Cl_f, \chi_f]$, where χ_f is χ expressed on the minimal ray class group, whose modulus is the conductor.

If flag = 2, $[f, bnr(f), \chi_f]$.

Note. Using this function with flag! = 0 is usually a bad idea and kept for compatibility and convenience only: flag = 1 has always been useless, since it is no faster than flag = 2 and returns less information; flag = 2 is mostly OK with two subtle drawbacks:

- it returns the full *bnr* attached to the full ray class group, whereas in applications we only need Cl_f modulo N-th powers, where N is any multiple of the exponent of Cl_f/H . Computing directly the conductor, then calling bnrinit with optional argument N avoids this problem.
- computing the *bnr* needs only be done once for each conductor, which is not possible using this function.

For maximal efficiency, the recommended procedure is as follows. Starting from data (character or congruence subgroups) attached to a modulus m, we can first compute the conductors using this function with default flag = 0. Then for all data with a common conductor f || m, compute (once!) the *bnr* attached to f using bnrinit (modulo N-th powers for a suitable N!) and finally map original data to the new *bnr* using bnrmap.

bnrconductorofchar(chi)

This function is obsolete, use bnrconductor.

bnrdisc(B, C, flag)

A, B, C defining a class field L over a ground field K (of type [:emphasis:`bnr]`, [:emphasis:`bnr, subgroup]`, [:emphasis:`bnr, character]`, [:emphasis:`bnf, modulus]` or [:emphasis:`bnf, modulus, subgroup]`, CFT (in the PARI manual)), outputs data $[N, r_1, D]$ giving the discriminant and signature of L, depending on the binary digits of flag:

- 1: if this bit is unset, output absolute data related to L/Q: N is the absolute degree [L : Q], r₁ the number of real places of L, and D the discriminant of L/Q. Otherwise, output relative data for L/K: N is the relative degree [L : K], r₁ is the number of real places of K unramified in L (so that the number of real places of L is equal to r₁ times N), and D is the relative discriminant ideal of L/K.
- 2: if this bit is set and if the modulus is not the conductor of L, only return 0.

bnrdisclist(bound, arch)

bnf being as output by **bnfinit** (with units), computes a list of discriminants of Abelian extensions of the number field by increasing modulus norm up to bound *bound*. The ramified Archimedean places are given by *arch*; all possible values are taken if *arch* is omitted.

The alternative syntax bnrdisclist(bnf, list) is supported, where *list* is as output by ideallist or ideallistarch (with units), in which case *arch* is disregarded.

The output v is a vector, where v[k] is itself a vector w, whose length is the number of ideals of norm k.

• We consider first the case where *arch* was specified. Each component of w corresponds to an ideal m of norm k, and gives invariants attached to the ray class field L of bnf of conductor [m, arch]. Namely, each

contains a vector [m, d, r, D] with the following meaning: m is the prime ideal factorization of the modulus, $d = [L : \mathbb{Q}]$ is the absolute degree of L, r is the number of real places of L, and D is the factorization of its absolute discriminant. We set d = r = D = 0 if m is not the finite part of a conductor.

• If arch was omitted, all $t = 2^{r_1}$ possible values are taken and a component of w has the form $[m, [[d_1, r_1, D_1], ..., [d_t, r_t, D_t]]]$, where m is the finite part of the conductor as above, and $[d_i, r_i, D_i]$ are the invariants of the ray class field of conductor $[m, v_i]$, where v_i is the *i*-th Archimedean component, ordered by inverse lexicographic order; so $v_1 = [0, ..., 0]$, $v_2 = [1, 0..., 0]$, etc. Again, we set $d_i = r_i = D_i = 0$ if $[m, v_i]$ is not a conductor.

Finally, each prime ideal $pr = [p, \alpha, e, f, \beta]$ in the prime factorization m is coded as the integer $p.n^2 + (f - 1).n + (j - 1)$, where n is the degree of the base field and j is such that

```
pr = idealprimedec(:emphasis:`nf,p)[j]`.
```

m can be decoded using **bnfdecodemodule**.

Note that to compute such data for a single field, either bnrclassno or bnrdisc are (much) more efficient.

bnrgaloisapply(mat, H)

Apply the automorphism given by its matrix *mat* to the congruence subgroup H given as a HNF matrix. The matrix *mat* can be computed with bnrgaloismatrix.

bnrgaloismatrix(aut)

Return the matrix of the action of the automorphism *aut* of the base field bnf.nf on the generators of the ray class field bnr.gen; *aut* can be given as a polynomial, an algebraic number, or a vector of automorphisms or a Galois group as output by galoisinit, in which case a vector of matrices is returned (in the later case, only for the generators aut.gen).

The generators bnr.gen need not be explicitly computed in the input *bnr*, which saves time: the result is well defined in this case also.

```
? K = bnfinit(a<sup>4</sup>-3*a<sup>2</sup>+253009); B = bnrinit(K,9); B.cyc
\%1 = [8400, 12, 6, 3]
? G = nfgaloisconj(K)
%2 = [-a, a, -1/503*a^3 + 3/503*a, 1/503*a^3 - 3/503*a]~
? bnrgaloismatrix(B, G[2]) \\ G[2] = Id ...
%3 =
[1 0 0 0]
[0 1 0 0]
[0 \ 0 \ 1 \ 0]
[0 0 0 1]
? bnrgaloismatrix(B, G[3]) \\ automorphism of order 2
%4 =
[799 0 0 2800]
[0704]
[4052]
[0002]
? M = %^2; for (i=1, #B.cyc, M[i,] %= B.cyc[i]); M
\%5 = \setminus  acts on ray class group as automorphism of order 2
[1 0 0 0]
```

[0 1 0 0] [0 0 1 0]

 $[0 \ 0 \ 0 \ 1]$

See bnrisgalois for further examples.

bnrinit(f, flag, cycmod)

bnf is as output by bnfinit (including fundamental units), f is a modulus, initializes data linked to the ray class group structure corresponding to this module, a so-called bnr structure. One can input the attached *bid* with generators for f instead of the module itself, saving some time. (As in idealstar, the finite part of the conductor may be given by a factorization into prime ideals, as produced by idealfactor.)

If the positive integer cycmod is present, only compute the ray class group modulo cycmod, which may save a lot of time when some maximal ideals in f have a huge residue field. In applications, we are given a congruence subgroup H and study the class field attached to Cl_f/H . If that finite Abelian group has an exponent which divides cycmod, then we have changed nothing theoretically, while trivializing expensive discrete logs in residue fields (since computations can be made modulo cycmod-th powers). This is useful in bnrclassfield, for instance when computing p-elementary extensions.

The following member functions are available on the result: .bnf is the underlying *bnf*, .mod the modulus, .bid the bid structure attached to the modulus; finally, .clgp, .no, .cyc, .gen refer to the ray class group (as a finite abelian group), its cardinality, its elementary divisors, its generators (only computed if flag = 1).

The last group of functions are different from the members of the underlying *bnf*, which refer to the class group; use :emphasis:`bnr.bnf.:emphasis:xxx`to access these, e.g. :emphasis:`bnr.bnf.cyc` to get the cyclic decomposition of the class group.

They are also different from the members of the underlying *bid*, which refer to $(\mathbb{Z}_K/f)^*$; use :emphasis:`bnr.bid.:emphasis::xxx`to access these, e.g. :emphasis:`bnr.bid.no`to get $\phi(f)$.

If flag = 0 (default), the generators of the ray class group are not explicitly computed, which saves time. Hence :emphasis: `bnr.gen` would produce an error. Note that implicit generators are still fixed and stored in the *bnr* (and guaranteed to be the same for fixed *bnf* and *bid* inputs), in terms of bnr.bnf.gen and bnr.bid.gen. The computation which is not performed is the expansion of such products in the ray class group so as to fix eplicit ideal representatives.

If flag = 1, as the default, except that generators are computed.

bnrisconductor(B, C)

Fast variant of bnrconductor(A, B, C); A, B, C represent an extension of the base field, given by class field theory (see CFT (in the PARI manual)). Outputs 1 if this modulus is the conductor, and 0 otherwise. This is slightly faster than bnrconductor when the character or subgroup is not primitive.

bnrisgalois(gal, H)

Check whether the class field attached to the subgroup H is Galois over the subfield of bnr.nf fixed by the group gal, which can be given as output by galoisinit, or as a matrix or a vector of matrices as output by bnrgaloismatrix, the second option being preferable, since it saves the recomputation of the matrices. Note: The function assumes that the ray class field attached to bnr is Galois, which is not checked.

In the following example, we lists the congruence subgroups of subextension of degree at most 3 of the ray class field of conductor 9 which are Galois over the rationals.

? K = bnfinit(a^4-3*a^2+253009); B = bnrinit(K,9); G = galoisinit(K); ? [H | H<-subgrouplist(B,3), bnrisgalois(B,G,H)]; time = 160 ms. ? M = bnrgaloismatrix(B,G); ? [H | H<-subgrouplist(B,3), bnrisgalois(B,M,H)] time = 1 ms.

The second computation is much faster since bnrgaloismatrix(B,G) is computed only once.

bnrisprincipal(x, flag)

Let *bnr* be the ray class group data output by $\mathtt{bnrinit}(,, 1)$ and let x be an ideal in any form, coprime to the modulus f = bnr.mod. Solves the discrete logarithm problem in the ray class group, with respect to the generators bnr.gen, in a way similar to bnfisprincipal. If x is not coprime to the modulus of *bnr* the result is undefined. Note that *bnr* need not contain the ray class group generators, i.e. it may be created with $\mathtt{bnrinit}(,,0)$; in that case, although $\mathtt{bnr.gen}$ is undefined, we can still fix natural generators for the ray class group (in terms of the generators in $\mathtt{bnr.bnf.gen}$ and $\mathtt{bnr.bid.gen}$) and compute with respect to them.

The binary digits of flag (default flag = 1) mean:

- 1: If set returns a 2-component vector [e, α] where e is the vector of components of x on the ray class group generators, α is an element congruent to 1 mod*f such that x = α Π_i g_i^{e_i}. If unset, returns only e.
- 4: If set, returns $[e, \alpha]$ where α is given in factored form (compact representation). This is orders of magnitude faster.

```
? K = bnfinit(x^2 - 30); bnr = bnrinit(K, [4, [1,1]]);
? bnr.clgp \\ ray class group is isomorphic to Z/4 x Z/2 x Z/2
%2 = [16, [4, 2, 2]]
? P = idealprimedec(K, 3)[1]; \\ the ramified prime ideal above 3
? bnrisprincipal(bnr,P) \\ bnr.gen undefined !
%5 = [[3, 0, 0]~, 9]
? bnrisprincipal(bnr,P, 0) \\ omit principal part
%5 = [3, 0, 0]~
? bnr = bnrinit(bnr, bnr.bid, 1); \\ include explicit generators
? bnrisprincipal(bnr,P) \\ ... alpha is different !
%7 = [[3, 0, 0]~, 1/128625]
```

It may be surprising that the generator α is different although the underlying *bnf* and *bid* are the same. This defines unique generators for the ray class group as ideal *classes*, whether we use bnrinit(,0) or bnrinit(,1). But the actual ideal representatives (implicit if the flag is 0, computed and stored in the *bnr* if the flag is 1) are in general different and this is what happens here. Indeed, the implicit generators are naturally expressed expressed in terms of bnr.bnf.gen and bnr.bid.gen and *then* expanded and simplified (in the same ideal class) so that we obtain ideal representatives for bnr.gen which are as simple as possible. And indeed the quotient of the two α found is 1 modulo the conductor (and positive at the infinite places it contains), and this is the only guaranteed property.

Beware that, when bnr is generated using bnrinit(, cycmod), the results are given in Cl_f modulo cycmod-th powers:

```
? bnr2 = bnrinit(K, bnr.mod,, 2); \\ modulo squares
? bnr2.clgp
%9 = [8, [2, 2, 2]] \\ bnr.clgp tensored by Z/2Z
? bnrisprincipal(bnr2,P, 0)
%10 = [1, 0, 0]~
```

bnrmap(B)

This function has two different uses:

• if A and B are *bnr* structures for the same *bnf* attached to moduli m_A and m_B with $m_B || m_A$, return the canonical surjection from A to B, i.e. from the ray class group modulo m_A to the ray class group modulo m_B . The map is coded by a triple $[M, cyc_A, cyc_B]$: M gives the image of the fixed ray class group generators of A in terms of the ones in B, cyc_A and cyc_B are the cyclic structures A.cyc and B.cyc respectively. Note that this function does *not* need A or B to contain explicit generators for the ray class groups: they may be created using bnrinit(,0).

If B is only known modulo N-th powers (from bnrinit(,N)), the result is correct provided N is a multiple of the exponent of A.

• if A is a projection map as above and B is either a congruence subgroup H, or a ray class character χ , or a discrete logarithm (from bnrisprincipal) modulo m_A whose conductor divides m_B , return the image of the subgroup (resp. the character, the discrete logarighm) as defined modulo m_B . The main use of this variant is to compute the primitive subgroup or character attached to a *bnr* modulo their conductor. This is more efficient than bnrconductor in two respects: the *bnr* attached to the conductor need only be computed once and, most importantly, the ray class group can be computed modulo N-th powers, where N is a multiple of the exponent of Cl_{m_A}/H (resp. of the order of χ). Whereas bnrconductor is specified to return a *bnr* attached to the full ray class group, which may lead to untractable discrete logarithms in the full ray class group instead of a tiny quotient.

bnrrootnumber(chi, flag, precision)

If $\chi = chi$ is a character over *bnr*, not necessarily primitive, let $L(s, \chi) = \sum_{id} \chi(id)N(id)^{-s}$ be the attached Artin L-function. Returns the so-called Artin root number, i.e. the complex number $W(\chi)$ of modulus 1 such that

$$\Lambda(1-s,\chi) = W(\chi)\Lambda(s,\overline{\chi})$$

where $\Lambda(s,\chi) = A(\chi)^{s/2} \gamma_{\chi}(s) L(s,\chi)$ is the enlarged L-function attached to L.

You can set flag = 1 if the character is known to be primitive. Example:

bnf = bnfinit(x^2 - x - 57); bnr = bnrinit(bnf, [7,[1,1]]); bnrrootnumber(bnr, [2,1])

returns the root number of the character χ of $\operatorname{Cl}_{7oo_1oo_2}(\mathbb{Q}(\sqrt{229}))$ defined by $\chi(g_1^a g_2^b) = \zeta_1^{2a} \zeta_2^b$. Here g_1, g_2 are the generators of the ray-class group given by bnr.gen and $\zeta_1 = e^{2i\pi/N_1}, \zeta_2 = e^{2i\pi/N_2}$ where N_1, N_2 are the orders of g_1 and g_2 respectively ($N_1 = 6$ and $N_2 = 3$ as bnr.cyc readily tells us).

bnrstark(subgroup, precision)

bnr being as output by **bnrinit**, finds a relative equation for the class field corresponding to the modulus in *bnr* and the given congruence subgroup (as usual, omit *subgroup* if you want the whole ray class group).

The main variable of *bnr* must not be x, and the ground field and the class field must be totally real. When the base field is \mathbb{Q} , the vastly simpler galoissubcyclo is used instead. Here is an example:

```
bnf = bnfinit(y^2 - 3);
bnr = bnrinit(bnf, 5);
bnrstark(bnr)
```

returns the ray class field of $\mathbb{Q}(\sqrt{3})$ modulo 5. Usually, one wants to apply to the result one of

```
rnfpolredbest(bnf, pol) \\ compute a reduced relative polynomial
rnfpolredbest(bnf, pol, 2) \\ compute a reduced absolute polynomial
```

The routine uses Stark units and needs to find a suitable auxiliary conductor, which may not exist when the class field is not cyclic over the base. In this case bnrstark is allowed to return a vector of polynomials defining *independent* relative extensions, whose compositum is the requested class field. We decided that it was useful to keep the extra information thus made available, hence the user has to take the compositum herself, see nfcompositum.

Even if it exists, the auxiliary conductor may be so large that later computations become unfeasible. (And of course, Stark's conjecture may simply be wrong.) In case of difficulties, try bnrclassfield:

```
? bnr = bnrinit(bnfinit(y^8-12*y^6+36*y^4-36*y^2+9,1), 2);
? bnrstark(bnr)
*** at top-level: bnrstark(bnr)
*** ^------
*** bnrstark: need 3919350809720744 coefficients in initzeta.
*** Computation impossible.
? bnrclassfield(bnr)
time = 20 ms.
%2 = [x^2 + (-2/3*y^6 + 7*y^4 - 14*y^2 + 3)]
```

call(A)

 $A = [a_1, ..., a_n]$ being a vector and f being a function, returns the evaluation of $f(a_1, ..., a_n)$. f can also be the name of a built-in GP function. If #A = 1, call (f, A) = apply (f, A)[1]. If f is variadic, the variadic arguments must grouped in a vector in the last component of A.

This function is useful

• when writing a variadic function, to call another one:

fprintf(file,format,args[..]) = write(file,call(strprintf,[format,args]))

· when dealing with function arguments with unspecified arity

The function below implements a global memoization interface:

```
memo=Map();
memoize(f,A[..])=
{
    my(res);
    if(!mapisdefined(memo, [f,A], &res),
    res = call(f,A);
    mapput(memo,[f,A],res));
    res;
}
```

for example:

```
? memoize(factor,2^128+1)
%3 = [59649589127497217,1;5704689200685129054721,1]
? ##
 *** last result computed in 76 ms.
? memoize(factor,2^128+1)
%4 = [59649589127497217,1;5704689200685129054721,1]
? ##
 *** last result computed in 0 ms.
? memoize(ffinit,3,3)
%5 = Mod(1,3)*x^3+Mod(1,3)*x^2+Mod(1,3)*x+Mod(2,3)
```

```
? fibo(n)=if(n==0,0,n==1,1,memoize(fibo,n-2)+memoize(fibo,n-1));
? fibo(100)
%7 = 354224848179261915075
```

• to call operators through their internal names without using alias

```
matnbelts(M) = call("_*_",matsize(M))
```

ceil()

Ceiling of x. When x is in \mathbb{R} , the result is the smallest integer greater than or equal to x. Applied to a rational function, ceil(x) returns the Euclidean quotient of the numerator by the denominator.

centerlift(v)

Same as lift, except that t_INTMOD and t_PADIC components are lifted using centered residues:

- for a t_INTMOD $x \in \mathbb{Z}/n\mathbb{Z}$, the lift y is such that -n/2 < y <= n/2.
- a t_PADIC x is lifted in the same way as above (modulo $p^padicprec(x)$) if its valuation v is nonnegative; if not, returns the fraction p^v centerlift (xp^{-v}) ; in particular, rational reconstruction is not attempted. Use bestappr for this.

For backward compatibility, centerlift(x, 'v) is allowed as an alias for lift(x, 'v).

characteristic()

Returns the characteristic of the base ring over which x is defined (as defined by t_INTMOD and t_FFELT components). The function raises an exception if incompatible primes arise from t_FFELT and t_PADIC components.

```
? characteristic(Mod(1,24)*x + Mod(1,18)*y)
%1 = 6
```

charconj(chi)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j \le k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the conjugate character.

```
? cyc = [15,5]; chi = [1,1];
? charconj(cyc, chi)
%2 = [14, 4]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charconj(bnf, [1])
%5 = [2]
```

For Dirichlet characters (when cyc is znstar(q,1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character:

```
? G = znstar(8, 1); \\ (Z/8Z)^*
? charorder(G, 3) \\ Conrey label
%2 = 2
? chi = znconreylog(G, 3);
```

? charorder(G, chi) \\ Conrey logarithm
%4 = 2

chardiv(a, b)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given two characters a and b, return the character $a/b = a\overline{b}$.

```
? cyc = [15,5]; a = [1,1]; b = [2,4];
? chardiv(cyc, a,b)
%2 = [14, 2]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? chardiv(bnf, [1], [2])
%5 = [2]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character. If the two characters are in the same format, the result is given in the same format, otherwise a Conrey logarithm is used.

```
? G = znstar(100, 1);
? G.cyc
%2 = [20, 2]
? a = [10, 1]; \\ usual representation for characters
? b = 7; \\ Conrey label;
? c = znconreylog(G, 11); \\ Conrey log
? chardiv(G, b,b)
%6 = 1 \\ Conrey label
? chardiv(G, a,b)
%7 = [0, 5]~ \\ Conrey log
%7 = [0, 14]~ \\ Conrey log
```

chareval(chi, x, z)

Let G be an abelian group structure affording a discrete logarithm method, e.g G = znstar(N, 1) for $(\mathbb{Z}/N\mathbb{Z})^*$ or a bnr structure, let x be an element of G and let chi be a character of G (see the note below for details). This function returns the value of chi at x.

Note on characters. Let K be some field. If G is an abelian group, let $\chi : G \to K^*$ be a character of finite order and let o be a multiple of the character order such that $\chi(n) = \zeta^{c(n)}$ for some fixed $\zeta \in K^*$ of multiplicative order o and a unique morphism $c : G \to (\mathbb{Z}/o\mathbb{Z}, +)$. Our usual convention is to write

$$G = (\mathbb{Z}/o_1\mathbb{Z})g_1 \oplus \ldots \oplus (\mathbb{Z}/o_d\mathbb{Z})g_d$$

for some generators (g_i) of respective order d_i , where the group has exponent $o := lcm_i o_i$. Since $\zeta^o = 1$, the vector (c_i) in $\prod (\mathbb{Z}/o_i\mathbb{Z})$ defines a character χ on G via $\chi(g_i) = \zeta^{c_i(o/o_i)}$ for all i. Classical Dirichlet characters have values in $K = \mathbb{C}$ and we can take $\zeta = \exp(2i\pi/o)$.

Note on Dirichlet characters. In the special case where *bid* is attached to $G = (\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), the Dirichlet character *chi* can be written in one of the usual 3 formats: a t_VEC in terms of bid.gen as

above, a t_COL in terms of the Conrey generators, or a t_INT (Conrey label); see dirichletchar (in the PARI manual) or ??character.

The character value is encoded as follows, depending on the optional argument z:

- If z is omitted: return the rational number c(x)/o for x coprime to q, where we normalize $0 \le c(x) \le o$. If x can not be mapped to the group (e.g. x is not coprime to the conductor of a Dirichlet or Hecke character) we return the sentinel value -1.
- If z is an integer o, then we assume that o is a multiple of the character order and we return the integer c(x) when x belongs to the group, and the sentinel value -1 otherwise.
- z can be of the form [zeta, o], where zeta is an o-th root of 1 and o is a multiple of the character order. We return $\zeta^{c(x)}$ if x belongs to the group, and the sentinel value 0 otherwise. (Note that this coincides with the usual extension of Dirichlet characters to \mathbb{Z} , or of Hecke characters to general ideals.)
- Finally, z can be of the form [vzeta, o], where vzeta is a vector of powers $\zeta^0, ..., \zeta^{o-1}$ of some o-th root of 1 and o is a multiple of the character order. As above, we return $\zeta^{c(x)}$ after a table lookup. Or the sentinel value 0.

chargalois(ORD)

Let *cyc* represent a finite abelian group by its elementary divisors (any object which has a .cyc method is also allowed, i.e. the output of znstar or bnrinit). Return a list of representatives for the Galois orbits of complex characters of G. If ORD is present, select characters depending on their orders:

- if ORD is a t_INT, restrict to orders less than this bound;
- if ORD is a t_VEC or t_VECSMALL, restrict to orders in the list.

```
? G = znstar(96);
? #chargalois(G) \\ 16 orbits of characters mod 96
%2 = 16
? #chargalois(G,4) \\ order less than 4
%3 = 12
? chargalois(G,[1,4]) \\ order 1 or 4; 5 orbits
%4 = [[0, 0, 0], [2, 0, 0], [2, 1, 0], [2, 0, 1], [2, 1, 1]]
```

Given a character χ , of order n (charorder(G, chi)), the elements in its orbit are the $\phi(n)$ characters χ^i , (i, n) = 1.

charker(chi)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the kernel of χ , as a matrix K in HNF which is a left-divisor of matdiagonal(d). Its columns express in terms of the g_i the generators of the subgroup. The determinant of K is the kernel index.

```
? cyc = [15,5]; chi = [1,1];
? charker(cyc, chi)
%2 =
[15 12]
[ 0 1]
? bnf = bnfinit(x^2+23);
? bnf.cyc
```

```
%4 = [3]
? charker(bnf, [1])
%5 =
[3]
```

Note that for Dirichlet characters (when cyc is znstar(q, 1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character.

? G = znstar(8, 1); \\ (Z/8Z)^*
? charker(G, 1) \\ Conrey label for trivial character
%2 =
[1 0]
[0 1]

charmul(a, b)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given two characters a and b, return the product character ab.

```
? cyc = [15,5]; a = [1,1]; b = [2,4];
? charmul(cyc, a,b)
%2 = [3, 0]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charmul(bnf, [1], [2])
%5 = [0]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character. If the two characters are in the same format, their product is given in the same format, otherwise a Conrey logarithm is used.

```
? G = znstar(100, 1);
? G.cyc
%2 = [20, 2]
? a = [10, 1]; \\ usual representation for characters
? b = 7; \\ Conrey label;
? c = znconreylog(G, 11); \\ Conrey log
? charmul(G, b,b)
%6 = 49 \\ Conrey label
? charmul(G, a,b)
%7 = [0, 15]~ \\ Conrey log
? charmul(G, a,c)
%7 = [0, 6]~ \\ Conrey log
```

charorder(chi)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $\chi = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

This function returns the order of the character chi.

```
? cyc = [15,5]; chi = [1,1];
? charorder(cyc, chi)
%2 = 15
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charorder(bnf, [1])
%5 = 3
```

For Dirichlet characters (when cyc is znstar(q, 1)), characters in Conrey representation are available, see dirichletchar (in the PARI manual) or ??character:

```
? G = znstar(100, 1); \\ (Z/100Z)^*
? charorder(G, 7) \\ Conrey label
%2 = 4
```

charpoly(*v*, *flag*)

characteristic polynomial of A with respect to the variable v, i.e. determinant of v * I - A if A is a square matrix.

? charpoly([1,2;3,4]); %1 = x^2 - 5*x - 2 ? charpoly([1,2;3,4],, 't) %2 = t^2 - 5*t - 2

If A is not a square matrix, the function returns the characteristic polynomial of the map "multiplication by A" if A is a scalar:

```
? charpoly(Mod(x+2, x^3-2))
%1 = x^3 - 6*x^2 + 12*x - 10
? charpoly(I)
%2 = x^2 + 1
? charpoly(quadgen(5))
%3 = x^2 - x - 1
? charpoly(ffgen(ffinit(2,4)))
%4 = Mod(1, 2)*x^4 + Mod(1, 2)*x^3 + Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2)
```

The value of flag is only significant for matrices, and we advise to stick to the default value. Let n be the dimension of A.

If flag = 0, same method (Le Verrier's) as for computing the adjoint matrix, i.e. using the traces of the powers of A. Assumes that n! is invertible; uses $O(n^4)$ scalar operations.

If flag = 1, uses Lagrange interpolation which is usually the slowest method. Assumes that n! is invertible; uses $O(n^4)$ scalar operations.

If flag = 2, uses the Hessenberg form. Assumes that the base ring is a field. Uses $O(n^3)$ scalar operations, but suffers from coefficient explosion unless the base field is finite or \mathbb{R} .

If flag = 3, uses Berkowitz's division free algorithm, valid over any ring (commutative, with unit). Uses $O(n^4)$ scalar operations.

If flag = 4, x must be integral. Uses a modular algorithm: Hessenberg form for various small primes, then Chinese remainders.

If flag = 5 (default), uses the "best" method given x. This means we use Berkowitz unless the base ring is \mathbb{Z} (use flag = 4) or a field where coefficient explosion does not occur, e.g. a finite field or the reals (use flag = 2).

charpow(a, n)

Let *cyc* represent a finite abelian group by its elementary divisors, i.e. (d_j) represents $\sum_{j < =k} \mathbb{Z}/d_j\mathbb{Z}$ with $d_k | ... || d_1$; any object which has a .cyc method is also allowed, e.g. the output of znstar or bnrinit. A character on this group is given by a row vector $a = [a_1, ..., a_n]$ such that $\chi(\prod g_j^{n_j}) = \exp(2\pi i \sum a_j n_j/d_j)$, where g_j denotes the generator (of order d_j) of the *j*-th cyclic component.

Given $n \in \mathbb{Z}$ and a character a, return the character a^n .

```
? cyc = [15,5]; a = [1,1];
? charpow(cyc, a, 3)
%2 = [3, 3]
? charpow(cyc, a, 5)
%2 = [5, 0]
? bnf = bnfinit(x^2+23);
? bnf.cyc
%4 = [3]
? charpow(bnf, [1], 3)
%5 = [0]
```

For Dirichlet characters on $(\mathbb{Z}/N\mathbb{Z})^*$, additional representations are available (Conrey labels, Conrey logarithm), see dirichletchar (in the PARI manual) or ??character and the output uses the same format as the input.

? G = znstar(100, 1); ? G.cyc %2 = [20, 2] ? a = [10, 1]; \\ standard representation for characters ? b = 7; \\ Conrey label; ? c = znconreylog(G, 11); \\ Conrey log ? charpow(G, a,3) %6 = [10, 1] \\ standard representation ? charpow(G, b,3) %7 = 43 \\ Conrey label ? charpow(G, c,3) %8 = [1, 8]~ \\ Conrey log

chinese(y)

If x and y are both intmods or both polmods, creates (with the same type) a z in the same residue class as x and in the same residue class as y, if it is possible.

? chinese(Mod(1,2), Mod(2,3))
%1 = Mod(5, 6)
? chinese(Mod(x,x^2-1), Mod(x+1,x^2+1))
%2 = Mod(-1/2*x^2 + x + 1/2, x^4 - 1)

This function also allows vector and matrix arguments, in which case the operation is recursively applied to each component of the vector or matrix.

```
? chinese([Mod(1,2),Mod(1,3)], [Mod(1,5),Mod(2,7)])
%3 = [Mod(1, 10), Mod(16, 21)]
```

For polynomial arguments in the same variable, the function is applied to each coefficient; if the polynomials have different degrees, the high degree terms are copied verbatim in the result, as if the missing high degree terms in the polynomial of lowest degree had been Mod(0, 1). Since the latter behavior is usually *not* the desired one, we propose to convert the polynomials to vectors of the same length first:

? P = x+1; Q = x²+2*x+1; ? chinese(P*Mod(1,2), Q*Mod(1,3)) %4 = Mod(1, 3)*x² + Mod(5, 6)*x + Mod(3, 6) ? chinese(Vec(P,3)*Mod(1,2), Vec(Q,3)*Mod(1,3)) %5 = [Mod(1, 6), Mod(5, 6), Mod(4, 6)] ? Pol(%) %6 = Mod(1, 6)*x² + Mod(5, 6)*x + Mod(4, 6)

If y is omitted, and x is a vector, chinese is applied recursively to the components of x, yielding a residue belonging to the same class as all components of x.

Finally chinese(x, x) = x regardless of the type of x; this allows vector arguments to contain other data, so long as they are identical in both vectors.

cmp(y)

Gives the result of a comparison between arbitrary objects x and y (as -1, 0 or 1). The underlying order relation is transitive, the function returns 0 if and only if x == y. It has no mathematical meaning but satisfies the following properties when comparing entries of the same type:

- two t_INT s compare as usual (i.e. cmp(x, y) < 0 if and only if x < y);
- two t_VECSMALL s of the same length compare lexicographically;
- two t_STR s compare lexicographically.

In case all components are equal up to the smallest length of the operands, the more complex is considered to be larger. More precisely, the longest is the largest; when lengths are equal, we have matrix > vector > scalar. For example:

? cmp(1, 2) %1 = -1 ? cmp(2, 1) %2 = 1 ? cmp(1, 1.0) \\ note that 1 == 1.0, but (1===1.0) is false. %3 = -1 ? cmp(x + Pi, []) %4 = -1

This function is mostly useful to handle sorted lists or vectors of arbitrary objects. For instance, if v is a vector, the construction vecsort(v, cmp) is equivalent to Set(v).

component(n)

Extracts the n - th-component of x. This is to be understood as follows: every PARI type has one or two initial code words. The components are counted, starting at 1, after these code words. In particular if x is a vector, this is indeed the n - th-component of x, if x is a matrix, the n - th column, if x is a polynomial, the n - th coefficient (i.e. of degree n - 1), and for power series, the n - th significant coefficient.

For polynomials and power series, one should rather use polcoeff, and for vectors and matrices, the [] operator. Namely, if x is a vector, then x[n] represents the n - th component of x. If x is a matrix, x[m,n] represents the coefficient of row m and column n of the matrix, x[m,] represents the m - th row of x, and x[,n] represents the n - th column of x.

Using of this function requires detailed knowledge of the structure of the different PARI types, and thus it should almost never be used directly. Some useful exceptions:

```
? x = 3 + 0(3<sup>5</sup>);
? component(x, 2)
```

```
%2 = 81 \\ p^(p-adic accuracy)
? component(x, 1)
%3 = 3 \\ p
? q = Qfb(1,2,3);
? component(q, 1)
%5 = 1
```

concat(y)

Concatenation of x and y. If x or y is not a vector or matrix, it is considered as a one-dimensional vector. All types are allowed for x and y, but the sizes must be compatible. Note that matrices are concatenated horizontally, i.e. the number of rows stays the same. Using transpositions, one can concatenate them vertically, but it is often simpler to use matconcat.

```
? x = matid(2); y = 2*matid(2);
? concat(x, y)
%2 =
[1 0 2 0]
[0 1 0 2]
? concat(x~,y~)~
%3 =
[1 0]
[0 1]
[2 0]
[0 2]
? matconcat([x;y])
%4 =
[1 0]
[0 1]
[2 0]
[0 2]
```

To concatenate vectors sideways (i.e. to obtain a two-row or two-column matrix), use Mat instead, or matconcat:

```
? x = [1,2];
? y = [3,4];
? concat(x,y)
%3 = [1, 2, 3, 4]
? Mat([x,y]~)
%4 =
[1 2]
[3 4]
? matconcat([x;y])
%5 =
```

[1 2] [3 4]

Concatenating a row vector to a matrix having the same number of columns will add the row to the matrix (top row if the vector is x, i.e. comes first, and bottom row otherwise).

The empty matrix [;] is considered to have a number of rows compatible with any operation, in particular concatenation. (Note that this is *not* the case for empty vectors [] or $[]\sim$.)

If y is omitted, x has to be a row vector or a list, in which case its elements are concatenated, from left to right, using the above rules.

```
? concat([1,2], [3,4])
%1 = [1, 2, 3, 4]
? a = [[1,2]~, [3,4]~]; concat(a)
%2 =
[1 3]
[2 4]
? concat([1,2; 3,4], [5,6]~)
%3 =
[1 2 5]
[3 4 6]
? concat([%, [7,8]~, [1,2,3,4]])
%5 =
[1 2 5 7]
[3 4 6 8]
[1 2 3 4]
```

conj()

Conjugate of x. The meaning of this is clear, except that for real quadratic numbers, it means conjugation in the real quadratic field. This function has no effect on integers, reals, intmods, fractions or p-adics. The only forbidden type is polmod (see conjvec for this).

conjvec(precision)

Conjugate vector representation of z. If z is a polmod, equal to Mod(a, T), this gives a vector of length degree(T) containing:

- the complex embeddings of z if T has rational coefficients, i.e. the a(r[i]) where r = polroots(T);
- the conjugates of z if T has some intmod coefficients;
- if z is a finite field element, the result is the vector of conjugates $[z, z^p, z^{p^2}, ..., z^{p^{n-1}}]$ where n = degree(T).

If z is an integer or a rational number, the result is z. If z is a (row or column) vector, the result is a matrix whose columns are the conjugate vectors of the individual elements of z.

content(D)

Computes the gcd of all the coefficients of x, when this gcd makes sense. This is the natural definition if x is a polynomial (and by extension a power series) or a vector/matrix. This is in general a weaker notion than the *ideal* generated by the coefficients:

? content(2*x+y)
%1 = 1 \\ = gcd(2,y) over Q[y]

If x is a scalar, this simply returns the absolute value of x if x is rational (t_INT or t_FRAC), and either 1 (inexact input) or x (exact input) otherwise; the result should be identical to gcd(x, 0).

The content of a rational function is the ratio of the contents of the numerator and the denominator. In recursive structures, if a matrix or vector *coefficient* x appears, the gcd is taken not with x, but with its content:

? content([[2], 4*matid(3)])
%1 = 2

The content of a t_VECSMALL is computed assuming the entries are signed integers.

The optional argument D allows to control over which ring we compute and get a more predictable behaviour:

- 1: we only consider the underlying \mathbb{Q} -structure and the denominator is a (positive) rational number
- a simple variable, say 'x: all entries are considered as rational functions in K(x) for some field K and the content is an element of K.

```
? f = x + 1/y + 1/2;
? content(f) \\ as a t_POL in x
%2 = 1/(2*y)
? content(f, 1) \\ Q-content
%3 = 1/2
? content(f, y) \\ as a rational function in y
%4 = 1/2
? g = x^2*y + y^2*x;
? content(g, x)
%6 = y
? content(g, y)
%7 = x
```

contfrac(b, nmax)

Returns the row vector whose components are the partial quotients of the continued fraction expansion of x. In other words, a result $[a_0, ..., a_n]$ means that $x a_0 + 1/(a_1 + ... + 1/a_n)$. The output is normalized so that $a_n! = 1$ (unless we also have n = 0).

The number of partial quotients n + 1 is limited by nmax. If nmax is omitted, the expansion stops at the last significant partial quotient.

```
? \p19
realprecision = 19 significant digits
? contfrac(Pi)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2]
? contfrac(Pi,, 3) \\ n = 2
%2 = [3, 7, 15]
```

x can also be a rational function or a power series.

If a vector b is supplied, the numerators are equal to the coefficients of b, instead of all equal to 1 as above; more precisely, $x (1/b_0)(a_0 + b_1/(a_1 + ... + b_n/a_n))$; for a numerical continued fraction (x real), the a_i are integers, as large as possible; if x is a rational function, they are polynomials with deg $a_i = \text{deg } b_i + 1$. The length of the result is then equal to the length of b, unless the next partial quotient cannot be reliably computed, in which case the expansion stops. This happens when a partial remainder is equal to zero (or too small compared to the available significant digits for x a t_REAL).

A direct implementation of the numerical continued fraction contfrac(x,b) described above would be

```
\\ "greedy" generalized continued fraction
cf(x, b) =
{ my( a= vector(#b), t );
    x *= b[1];
    for (i = 1, #b,
    a[i] = floor(x);
    t = x - a[i]; if (!t || i == #b, break);
    x = b[i+1] / t;
    ); a;
}
```

There is some degree of freedom when choosing the a_i ; the program above can easily be modified to derive variants of the standard algorithm. In the same vein, although no builtin function implements the related Engel expansion (a special kind of Egyptian fraction decomposition: $x = 1/a_1 + 1/(a_1a_2) + ...$), it can be obtained as follows:

```
\\ n terms of the Engel expansion of x
engel(x, n = 10) =
{ my( u = x, a = vector(n) );
for (k = 1, n,
    a[k] = ceil(1/u);
    u = u*a[k] - 1;
    if (!u, break);
    ); a
}
```

Obsolete hack. (don't use this): if b is an integer, *nmax* is ignored and the command is understood as contfrac(:math:`x,, b)`.

contfraceval(t, lim)

Given a continued fraction CF output by contfracinit, evaluate the first lim terms of the continued fraction at t (all terms if lim is negative or omitted; if positive, lim must be less than or equal to the length of CF.

contfracinit(lim)

Given M representing the power series $S = \sum_{n>=0} M[n+1]z^n$, transform it into a continued fraction in Euler form, using the quotient-difference algorithm; restrict to $n \le lim$ if latter is nonnegative. M can be a vector, a power series, a polynomial; if the limiting parameter lim is present, a rational function is also allowed (and converted to a power series of that accuracy).

The result is a 2-component vector [A, B] such that $S = M[1]/(1 + A[1]z + B[1]z^2/(1 + A[2]z + B[2]z^2/(1 + ...1/(1 + A[lim/2]z))))$. Does not work if any coefficient of M vanishes, nor for series for which certain partial denominators vanish.

contfracpnqn(n)

When x is a vector or a one-row matrix, x is considered as the list of partial quotients $[a_0, a_1, ..., a_n]$ of a rational number, and the result is the 2 by 2 matrix $[p_n, p_{n-1}; q_n, q_{n-1}]$ in the standard notation of continued fractions, so $p_n/q_n = a_0 + 1/(a_1 + ... + 1/a_n)$. If x is a matrix with two rows $[b_0, b_1, ..., b_n]$ and $[a_0, a_1, ..., a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = (1/b_0)(a_0+b_1/(a_1+...+b_n/a_n))$. Note that in this case one usually has $b_0 = 1$.

If $n \ge 0$ is present, returns all convergents from p_0/q_0 up to p_n/q_n . (All convergents if x is too small to compute the n + 1 requested convergents.)

```
? a = contfrac(Pi,10)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 3]
? allpnqn(x) = contfracpnqn(x,#x) \\ all convergents
? allpnqn(a)
%3 =
[3 22 333 355 103993 104348 208341 312689 1146408]
[1 7 106 113 33102 33215 66317 99532 364913]
? contfracpnqn(a) \\ last two convergents
%4 =
[1146408 312689]
[ 364913 99532]
? contfracpnqn(a,3) \\ first three convergents
%5 =
[3 22 333 355]
[1 7 106 113]
```

core(flag)

If n is an integer written as $n = df^2$ with d squarefree, returns d. If flag is nonzero, returns the two-element row vector [d, f]. By convention, we write $0 = 0x1^2$, so core(\emptyset , 1) returns [0, 1].

coredisc(flag)

A fundamental discriminant is an integer of the form $t = 1 \mod 4$ or $4t = 8, 12 \mod 16$, with t squarefree (i.e. 1 or the discriminant of a quadratic number field). Given a nonzero integer n, this routine returns the (unique) fundamental discriminant d such that $n = df^2$, f a positive rational number. If flag is nonzero, returns the two-element row vector [d, f]. If n is congruent to 0 or 1 modulo 4, f is an integer, and a half-integer otherwise.

By convention, coredisc(0, 1)) returns [0, 1].

Note that quaddisc(n) returns the same value as coredisc(n), and also works with rational inputs $n \in \mathbb{Q}^*$.

cos(*precision*)

Cosine of x. Note that, for real x, cosine and sine can be obtained simultaneously as

cs(x) = my(z = exp(I*x)); [real(z), imag(z)];

and for general complex x as

 $cs2(x) = my(z = exp(I^*x), u = 1/z); [(z+u)/2, (z-u)/2];$

Note that the latter function suffers from catastrophic cancellation when z^2 1.

cosh(precision)

Hyperbolic cosine of x.

cotan(precision)

Cotangent of x.

cotanh(precision)

Hyperbolic cotangent of x.

denominator(D)

Denominator of f. The meaning of this is clear when f is a rational number or function. If f is an integer or a polynomial, it is treated as a rational number or function, respectively, and the result is equal to 1. For polynomials, you probably want to use

denominator(content(f))

instead. As for modular objects, t_INTMOD and t_PADIC have denominator 1, and the denominator of a t_POLMOD is the denominator of its lift.

If f is a recursive structure, for instance a vector or matrix, the lcm of the denominators of its components (a common denominator) is computed. This also applies for t_COMPLEX s and t_QUAD s.

Warning. Multivariate objects are created according to variable priorities, with possibly surprising side effects (x/y) is a polynomial, but y/x is a rational function). See priority (in the PARI manual).

The optional argument D allows to control over which ring we compute the denominator and get a more predictable behaviour:

• 1: we only consider the underlying Q-structure and the denominator is a (positive) rational integer

• a simple variable, say 'x: all entries as rational functions in K(x) and the denominator is a polynomial in x.

```
? f = x + 1/y + 1/2;
? denominator(f) \\ a t_POL in x
%2 = 1
? denominator(f, 1) \\ Q-denominator
%3 = 2
? denominator(f, x) \\ as a t_POL in x, seen above
%4 = 1
? denominator(f, y) \\ as a rational function in y
%5 = 2*y
```

deriv(v)

Derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise. One can use x' as a shortcut if the derivative is with respect to the main variable of x; and also use x'', etc., for multiple derivatives altough derivn is often preferrable.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the derivative is taken with respect to the main variable used in the base ring R.

```
? f = (x/y)^5;
? deriv(f)
%2 = 5/y^5*x^4
? f'
%3 = 5/y^5*x^4
? deriv(f, 'x) \\ same since 'x is the main variable
%4 = 5/y^5*x^4
? deriv(f, 'y)
%5 = -5/y^6*x^5
```

This function also operates on closures, in which case the variable must be omitted. It returns a closure performing a numerical differentiation as per derivnum:

derivn(n, v)

n-th derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise; the integer n must be nonnegative. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise. One can use x', x, etc., as a shortcut if the derivative is with respect to the main variable of x.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the derivative is taken with respect to the main variable used in the base ring R.

? f = (x/y)^5; ? derivn(f, 2) %2 = 20/y^5*x^3 ? f'' %3 = 20/y^5*x^3 ? derivn(f, 2, 'x) \\ same since 'x is the main variable %4 = 20/y^5*x^3 ? derivn(f, 2, 'y) %5 = 30/y^7*x^5

This function also operates on closures, in which case the variable must be omitted. It returns a closure performing a numerical differentiation as per derivnum:

diffop(v, d, n)

Let v be a vector of variables, and d a vector of the same length, return the image of x by the n-power (1 if n is not given) of the differential operator D that assumes the value d[i] on the variable v[i]. The value of D on a scalar type is zero, and D applies componentwise to a vector or matrix. When applied to a t_POLMOD, if no value is provided for the variable of the modulus, such value is derived using the implicit function theorem.

Examples. This function can be used to differentiate formal expressions: if $E = \exp(X^2)$ then we have E' = 2 * X * E. We derivate $X * exp(X^2)$ as follows:

```
? diffop(E*X,[X,E],[1,2*X*E])
%1 = (2*X^2 + 1)*E
```

Let Sin and Cos be two function such that $Sin^2 + Cos^2 = 1$ and Cos' = -Sin. We can differentiate Sin/Cos as follows, PARI inferring the value of Sin' from the equation:

```
? diffop(Mod('Sin/'Cos,'Sin^2+'Cos^2-1),['Cos],[-'Sin])
%1 = Mod(1/Cos^2, Sin^2 + (Cos^2 - 1))
```

Compute the Bell polynomials (both complete and partial) via the Faa di Bruno formula:

```
Bell(k,n=-1)=
{ my(x, v, dv, var = i->eval(Str("X",i)));
v = vector(k, i, if (i==1, 'E, var(i-1)));
dv = vector(k, i, if (i==1, 'X*var(1)*'E, var(i)));
x = diffop('E,v,dv,k) / 'E;
if (n < 0, subst(x,'X,1), polcoef(x,n,'X));
}</pre>
```

digits(b)

Outputs the vector of the digits of ||x|| in base b, where x and b are integers (b = 10 by default). For $x \ge 1$, the number of digits is logint(x, b) + 1. See from digits for the reverse operation.

? digits(1230) %1 = [1, 2, 3, 0] ? digits(10, 2) \\ base 2 %2 = [1, 0, 1, 0]

By convention, 0 has no digits:

? digits(0) %3 = []

dilog(precision)

Principal branch of the dilogarithm of x, i.e. analytic continuation of the power series $\log_2(x) = \sum_{n>=1} x^n/n^2$.

dirdiv(y)

x and y being vectors of perhaps different lengths but with y[1]! = 0 considered as Dirichlet series, computes the quotient of x by y, again as a vector.

dirmul(y)

x and y being vectors of perhaps different lengths representing the Dirichlet series $\sum_n x_n n^{-s}$ and $\sum_n y_n n^{-s}$, computes the product of x by y, again as a vector.

? dirmul(vector(10,n,1), vector(10,n,moebius(n)))
%1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The product length is the minimum of # x * v(y) and # y * v(x), where v(x) is the index of the first nonzero coefficient.

? dirmul([0,1], [0,1]); %2 = [0, 0, 0, 1]

dirpowerssum(*x*, *precision*)

For positive integer n and complex number x, return the sum $1^x + 2^x + ... + n^x$. This is the same as vecsum(dirpowers(n,x)), but faster and using only $O(\sqrt{n})$ memory instead of O(n).

```
? dirpowers(5, 2)
%1 = [1, 4, 9, 16, 25]
? vecsum(%)
%2 = 55
? dirpowerssum(5, 2)
%3 = 55
? \p200
? dirpowerssum(10^7, 1/2 + I * sqrt(3));
time = 29,884 ms.
? vecsum(dirpowers(10^7, 1/2 + I * sqrt(3)))
time = 41,894 ms.
```

The penultimate command works with default stack size, the last one requires a stacksize of at least 5GB.

When $n \leq 0$, the function returns 0.

dirzetak(b)

Gives as a vector the first b coefficients of the Dedekind zeta function of the number field nf considered as a Dirichlet series.

divisors(flag)

Creates a row vector whose components are the divisors of x. The factorization of x (as output by factor) can be used instead. If flag = 1, return pairs [d, factor(d)].

By definition, these divisors are the products of the irreducible factors of n, as produced by factor(n), raised to appropriate powers (no negative exponent may occur in the factorization). If n is an integer, they are the positive divisors, in increasing order.

```
? divisors(12)
%1 = [1, 2, 3, 4, 6, 12]
? divisors(12, 1) \\ include their factorization
%2 = [[1, matrix(0,2)], [2, Mat([2, 1])], [3, Mat([3, 1])],
[4, Mat([2, 2])], [6, [2, 1; 3, 1]], [12, [2, 2; 3, 1]]]
? divisors(x<sup>4</sup> + 2*x<sup>3</sup> + x<sup>2</sup>) \\ also works for polynomials
%3 = [1, x, x<sup>2</sup>, x + 1, x<sup>2</sup> + x, x<sup>3</sup> + x<sup>2</sup>, x<sup>2</sup> + 2*x + 1,
x<sup>3</sup> + 2*x<sup>2</sup> + x, x<sup>4</sup> + 2*x<sup>3</sup> + x<sup>2</sup>]
```

This function requires a lot of memory if x has many divisors. The following idiom runs through all divisors using very little memory, in no particular order this time:

F = factor(x); P = F[,1]; E = F[,2]; forvec(e = vectorv(#E,i,[0,E[i]]), d = factorback(P,e); ...)

If the factorization of d is also desired, then [P, e] almost provides it but not quite: e may contain 0 exponents, which are not allowed in factorizations. These must be sieved out as in:

tofact(P,E) =
 my(v = select(x->x, E, 1)); Mat([vecextract(P,v), vecextract(E,v)]);
? tofact([2,3,5,7]~, [4,0,2,0]~)
%4 =
 [2 4]
 [5 2]

We can then run the above loop with tofact(P,e) instead of, or together with, factorback.

divisorslenstra(r, s)

Given three integers N > s > r >= 0 such that (r, s) = 1 and $s^3 > N$, find all divisors d of N such that d = r(mods). There are at most 11 such divisors (Lenstra).

? N = 245784; r = 19; s = 65; ? divisorslenstra(N, r, s) %2 = [19, 84, 539, 1254, 3724, 245784] ? [d | d <- divisors(N), d % s == r] %3 = [19, 84, 539, 1254, 3724, 245784]

When the preconditions are not met, the result is undefined:

```
? N = 4484075232; r = 7; s = 1303; s^3 > N
%4 = 0
? divisorslenstra(N, r, s)
? [ d | d <- divisors(N), d % s == r ]
%6 = [7, 2613, 9128, 19552, 264516, 3407352, 344928864]
```

(Divisors were missing but $s^3 < N$.)

divrem(y, v)

Creates a column vector with two components, the first being the Euclidean quotient (:math:`x \:math:y`), the second the Euclidean remainder (:math:`x - $(x\:math:y)^*:math:y`$), of the division of x by y. This avoids the need to do two divisions if one needs both the quotient and the remainder. If v is present, and x, y are multivariate polynomials, divide with respect to the variable v.

Beware that divrem(:math:`x,:math:y)[2]` is in general not the same as :math:`x % y; no GP operator corresponds to it:

eint1(n, precision)

Exponential integral $\int_x^o o(e^{-t})/(t)dt = incgam(0, x)$, where the latter expression extends the function definition from real x > 0 to all complex x! = 0.

If n is present, we must have x > 0; the function returns the n-dimensional vector [eint1(x), ..., eint1(nx)]. Contrary to other transcendental functions, and to the default case (n omitted), the values are correct up to a bounded *absolute*, rather than relative, error 10^{-n} , where n is precision(x) if x is a t_REAL and defaults to realprecision otherwise. (In the most important application, to the computation of L-functions via approximate functional equations, those values appear as weights in long sums and small individual relative errors are less useful than controlling the absolute error.) This is faster than repeatedly calling eint1(:math:`i * x)`, but less precise. ellE(precision)

Complete elliptic integral of the second kind

$$E(k) = \int_0^{\pi/2} (1 - k^2 \sin(t)^2)^{1/2} dt$$

for the complex parameter k using the agm.

ellK(precision)

Complete elliptic integral of the first kind

$$K(k) = \int_0^{\pi/2} (1 - k^2 \sin(t)^2)^{-1/2} dt$$

for the complex parameter k using the agm.

ellL1(r, precision)

Returns the value at s = 1 of the derivative of order r of the L-function of the elliptic curve E.

```
? E = ellinit("11a1"); \\ order of vanishing is 0
? ellL1(E)
%2 = 0.2538418608559106843377589233
? E = ellinit("389a1"); \\ order of vanishing is 2
? ellL1(E)
%4 = -5.384067311837218089235032414 E-29
? ellL1(E, 1)
%5 = 0
? ellL1(E, 2)
%6 = 1.518633000576853540460385214
```

The main use of this function, after computing at *low* accuracy the order of vanishing using ellanalyticrank, is to compute the leading term at *high* accuracy to check (or use) the Birch and Swinnerton-Dyer conjecture:

```
? \p18
realprecision = 18 significant digits
? E = ellinit("5077a1"); ellanalyticrank(E)
time = 8 ms.
%1 = [3, 10.3910994007158041]
? \p200
realprecision = 202 significant digits (200 digits displayed)
? ellL1(E, 3)
time = 104 ms.
%3 = 10.3910994007158041387518505103609170697263563756570092797[...]
```

elladd(*z*1, *z*2)

Sum of the points z1 and z2 on the elliptic curve corresponding to E.

ellak(n)

Computes the coefficient a_n of the *L*-function of the elliptic curve E/\mathbb{Q} , i.e. coefficients of a newform of weight 2 by the modularity theorem (Taniyama-Shimura-Weil conjecture). E must be an ell structure over \mathbb{Q} as output by elliptic. E must be given by an integral model, not necessarily minimal, although a minimal model will make the function faster.

? E = ellinit([1,-1,0,4,3]); ? ellak(E, 10)

```
%2 = -3
? e = ellchangecurve(E, [1/5,0,0,0]); \\ made not minimal at 5
? ellak(e, 10) \\ wasteful but works
%3 = -3
? E = ellminimalmodel(e); \\ now minimal
? ellak(E, 5)
%5 = -3
```

If the model is not minimal at a number of bad primes, then the function will be slower on those n divisible by the bad primes. The speed should be comparable for other n:

```
? for(i=1,10^6, ellak(E,5))
time = 699 ms.
? for(i=1,10^6, ellak(e,5)) \\ 5 is bad, markedly slower
time = 1,079 ms.
? for(i=1,10^5,ellak(E,5*i))
time = 1,477 ms.
? for(i=1,10^5,ellak(e,5*i)) \\ still slower but not so much on average
time = 1,569 ms.
```

ellan(n)

Computes the vector of the first *n* Fourier coefficients a_k corresponding to the elliptic curve *E* defined over a number field. If *E* is defined over \mathbb{Q} , the curve may be given by an arbitrary model, not necessarily minimal, although a minimal model will make the function faster. Over a more general number field, the model must be locally minimal at all primes above 2 and 3.

ellanalyticrank(eps, precision)

Returns the order of vanishing at s = 1 of the *L*-function of the elliptic curve *E* and the value of the first nonzero derivative. To determine this order, it is assumed that any value less than eps is zero. If eps is omitted, $2^{-b/2}$ is used, where *b* is the current bit precision.

```
? E = ellinit("11a1"); \\ rank 0
? ellanalyticrank(E)
%2 = [0, 0.2538418608559106843377589233]
? E = ellinit("37a1"); \\ rank 1
? ellanalyticrank(E)
%4 = [1, 0.3059997738340523018204836835]
? E = ellinit("389a1"); \\ rank 2
? ellanalyticrank(E)
%6 = [2, 1.518633000576853540460385214]
? E = ellinit("5077a1"); \\ rank 3
? ellanalyticrank(E)
%8 = [3, 10.39109940071580413875185035]
```

ellap(p)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. If the field $K = \mathbb{F}_q$ is finite, return the trace of Frobenius t, defined by the equation $\#E(\mathbb{F}_q) = q + 1 - t$.

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the trace of Frobenius for the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number $(K = \mathbb{Q})$ or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The equation need not be minimal or even integral at p; of course, a minimal model will be more efficient.

For a number field K, the trace of Frobenius is the a_p coefficient in the Euler product defining the curve L-series, whence the function name:

$$L(E/K,s) = \prod_{badp} (1 - a_p (Np)^{-s})^{-1} \prod_{goodp} (1 - a_p (Np)^{-s} + (Np)^{1-2s})^{-1}$$

When the characteristic of the finite field is large, the availability of the seadata package will speed up the computation.

```
? E = ellinit([0,1]); \ y^2 = x^3 + 0.x + 1, defined over Q
? ellap(E, 7) \setminus 7 necessary here
\%2 = -4 \setminus \#E(F_7) = 7+1-(-4) = 12
? ellcard(E, 7)
%3 = 12 \\ OK
? E = ellinit([0,1], 11); \setminus defined over F_{11}
? ellap(E) \setminus no need to repeat 11
%4 = 0
? ellap(E, 11) \setminus ... but it also works
\%5 = 0
? ellgroup(E, 13) \\ ouch, inconsistent input!
*** at top-level: ellap(E,13)
 *** ellap: inconsistent moduli in Rg_to_Fp:
11
13
? a = ffgen(ffinit(11,3), 'a); \\ defines F_q := F_{11^3}
? E = ellinit([a+1,a]); \ y^2 = x^3 + (a+1)x + a, defined over F_q
? ellap(E)
%8 = -3
```

If the curve is defined over a more general number field than \mathbb{Q} , the maximal ideal p must be explicitly given in idealprimedec format. There is no assumption of local minimality at p.

```
? K = nfinit(a^{2+1}); E = ellinit([1+a,0,1,0,0], K);
? fa = idealfactor(K, E.disc)
%2 =
[ [5, [-2, 1], 1, 1, [2, -1; 1, 2] ] 1 ]
[[13, [5, 1]~, 1, 1, [-5, -1; 1, -5]] 2]
? ellap(E, fa[1,1])
\%3 = -1 \setminus \text{nonsplit multiplicative reduction}
? ellap(E, fa[2,1])
%4 = 1 \\ split multiplicative reduction
? P17 = idealprimedec(K, 17)[1];
? ellap(E, P17)
\%6 = 6 \setminus \text{good reduction}
? E2 = ellchangecurve(E, [17,0,0,0]);
? ellap(E2, P17)
\% = 6 \\ same, starting from a nonmiminal model
? P3 = idealprimedec(K,3)[1];
? ellap(E, P3) \\ OK: E is minimal at P3
\%10 = -2
? E3 = ellchangecurve(E, [3,0,0,0]);
```

```
? ellap(E3, P3) \\ not integral at P3
 *** at top-level: ellap(E3,P3)
 *** ^------
 *** ellap: impossible inverse in Rg_to_ff: Mod(0, 3).
```

Algorithms used. If E/\mathbb{F}_q has CM by a principal imaginary quadratic order we use a fast explicit formula (involving essentially Kronecker symbols and Cornacchia's algorithm), in $O(\log q)^2$ bit operations. Otherwise, we use Shanks-Mestre's baby-step/giant-step method, which runs in time $O(q^{1/4})$ using $O(q^{1/4})$ storage, hence becomes unreasonable when q has about 30 digits. Above this range, the SEA algorithm becomes available, heuristically in $O(\log q)^4$, and primes of the order of 200 digits become feasible. In small characteristic we use Mestre's (p = 2), Kohel's (p = 3,5,7,13), Satoh-Harley (all in $O(p^2n^2)$) or Kedlaya's (in $O(pn^3)$) algorithms.

ellbil(z1, z2, precision)

Deprecated alias for ellheight(E,P,Q).

ellbsd(precision)

The object E being an elliptic curve over a number field, returns a real number c such that the BSD conjecture predicts that $L_E^{(r)}(1)/r! = cRS$ where r is the rank, R the regulator and S the cardinal of the Tate-Shafarevich group.

```
? e = ellinit([0,-1,1,-10,-20]); \\ rank 0
? ellbsd(e)
\%2 = 0.25384186085591068433775892335090946105
? lfun(e,1)
%3 = 0.25384186085591068433775892335090946104
? e = ellinit([0,0,1,-1,0]); \setminus rank 1
? P = ellheegner(e);
? ellbsd(e)*ellheight(e,P)
%6 = 0.30599977383405230182048368332167647445
? lfun(e,1,1)
%7 = 0.30599977383405230182048368332167647445
? e = ellinit([1+a,0,1,0,0],nfinit(a^2+1)); \\ rank 0
? ellbsd(e)
\%9 = 0.42521832235345764503001271536611593310
? lfun(e,1)
%10 = 0.42521832235345764503001271536611593309
```

ellcard(p)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. If $K = \mathbb{F}_q$ is finite, return the order of the group $E(\mathbb{F}_q)$.

? E = ellinit([-3,1], 5); ellcard(E) %1 = 7 ? t = ffgen(3^5,'t); E = ellinit([t,t^2+1]); ellcard(E) %2 = 217

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the order of the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number ($K = \mathbb{Q}$) or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The equation need not be minimal or even integral at p; of course, a minimal model will be more efficient. The function considers the group of nonsingular points of the reduction of a minimal model of the curve at p, so also makes sense when the curve has bad reduction.

```
? E = ellinit([-3,1]);
? factor(E.disc)
%2 =
[2 4]
[3 4]
? ellcard(E, 5) \\ as above !
%3 = 7
? ellcard(E, 2) \\ additive reduction
%4 = 2
```

When the characteristic of the finite field is large, the availability of the seadata package will speed the computation. See also ellap for the list of implemented algorithms.

ellchangecurve(v)

Changes the data for the elliptic curve E by changing the coordinates using the vector $\mathbf{v} = [\mathbf{u}, \mathbf{r}, \mathbf{s}, \mathbf{t}]$, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. E must be an ell structure as output by ellinit. The special case v = 1 is also used instead of [1, 0, 0, 0] to denote the trivial coordinate change.

ellchangepoint(v)

Changes the coordinates of the point or vector of points x using the vector v = [u, r, s, t], i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (see also ellchangecurve).

? E0 = ellinit([1,1]); P0 = [0,1]; v = [1,2,3,4]; ? E = ellchangecurve(E0, v); ? P = ellchangepoint(P0,v) %3 = [-2, 3] ? ellisoncurve(E, P) %4 = 1 ? ellchangepointinv(P,v) %5 = [0, 1]

ellchangepointinv(v)

Changes the coordinates of the point or vector of points x using the inverse of the isomorphism attached to v = [u,r,s,t], i.e. if x' and y' are the old coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (inverse of ellchangepoint).

```
? E0 = ellinit([1,1]); P0 = [0,1]; v = [1,2,3,4];
? E = ellchangecurve(E0, v);
? P = ellchangepoint(P0,v)
%3 = [-2, 3]
? ellisoncurve(E, P)
%4 = 1
? ellchangepointinv(P,v)
%5 = [0, 1] \\ we get back P0
```

ellconvertname()

Converts an elliptic curve name, as found in the elldata database, from a string to a triplet [conductor, isogenyclass, index]. It will also convert a triplet back to a curve name. Examples:

```
? ellconvertname("123b1")
%1 = [123, 1, 1]
? ellconvertname(%)
%2 = "123b1"
```
elldivpol(n, v)

n-division polynomial f_n for the curve E in the variable v. In standard notation, for any affine point P = (X, Y) on the curve and any integer $n \ge 0$, we have

$$[n]P = (\phi_n(P)\psi_n(P) : \omega_n(P) : \psi_n(P)^3)$$

for some polynomials ϕ_n, ω_n, ψ_n in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][X, Y]$. We have $f_n(X) = \psi_n(X)$ for n odd, and $f_n(X) = \psi_n(X, Y)(2Y + a_1X + a_3)$ for n even. We have

$$f_0 = 0, f_1 = 1, f_2 = 4X^3 + b_2X^2 + 2b_4X + b_6, f_3 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_3 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_3 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b_8, f_8 = 3X^4 + b_2X^3 + b_2X^3 + b_2X^2 + b_2X^3 + b_2X^2 + b_3X^2 + b_3X^2$$

$$f_4 = f_2(2X^6 + b_2X^5 + 5b_4X^4 + 10b_6X^3 + 10b_8X^2 + (b_2b_8 - b_4b_6)X + (b_8b_4 - b_6^2)), \dots$$

When n is odd, the roots of f_n are the X-coordinates of the affine points in the n-torsion subgroup E[n]; when n is even, the roots of f_n are the X-coordinates of the affine points in E[n] E[2] when n > 2, resp. in E[2] when n = 2. For n < 0, we define $f_n := -f_{-n}$.

elleisnum(k, flag, precision)

k being an even positive integer, computes the numerical value of the Eisenstein series of weight k at the lattice w, as given by ellperiods, namely

$$(2i\pi/\omega_2)^k(1+2/\zeta(1-k)\sum_{n>=1}n^{k-1}q^n/(1-q^n)),$$

where $q = \exp(2i\pi\tau)$ and $\tau := \omega_1/\omega_2$ belongs to the complex upper half-plane. It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by ellinit.

When *flag* is nonzero and k = 4 or 6, returns the elliptic invariants g_2 or g_3 , such that

$$y^2 = 4x^3 - g_2x - g_3$$

is a Weierstrass equation for E.

elleta(precision)

Returns the quasi-periods $[\eta_1, \eta_2]$ attached to the lattice basis $w = [\omega_1, \omega_2]$. Alternatively, w can be an elliptic curve E as output by ellinit, in which case, the quasi periods attached to the period lattice basis :math: `E.omega` (namely, :math: `E.eta`) are returned.

? elleta([1, I]) %1 = [3.141592653589793238462643383, 9.424777960769379715387930149*I]

ellformaldifferential(serprec, n)

Let $\omega := dx/(2y + a_1x + a_3)$ be the invariant differential form attached to the model E of some elliptic curve (ellinit form), and $\eta := x(t)\omega$. Return n terms (seriesprecision by default) of f(t), g(t) two power series in the formal parameter t = -x/y such that $\omega = f(t)dt, \eta = g(t)dt$:

$$f(t) = 1 + a_1 t + (a_1^2 + a_2)t^2 + \dots, g(t) = t^{-2} + \dots$$

? E = ellinit([-1,1/4]); [f,g] = ellformaldifferential(E,7,'t); ? f %2 = 1 - 2*t^4 + 3/4*t^6 + 0(t^7) ? g %3 = t^-2 - t^2 + 1/2*t^4 + 0(t^5)

ellformalexp(serprec, n)

The elliptic formal exponential Exp attached to E is the isomorphism from the formal additive law to the formal group of E. It is normalized so as to be the inverse of the elliptic logarithm (see ellformallog): ExpoL = Id. Return n terms of this power series:

? E=ellinit([-1,1/4]); Exp = ellformalexp(E,10,'z) %1 = z + 2/5*z^5 - 3/28*z^7 + 2/15*z^9 + 0(z^11) ? L = ellformallog(E,10,'t); ? subst(Exp,z,L) %3 = t + 0(t^11)

ellformallog(serprec, n)

The formal elliptic logarithm is a series L in tK[[t]] such that $dL = \omega = dx/(2y + a_1x + a_3)$, the canonical invariant differential attached to the model E. It gives an isomorphism from the formal group of E to the additive formal group.

? E = ellinit([-1,1/4]); L = ellformallog(E, 9, 't) %1 = t - 2/5*t^5 + 3/28*t^7 + 2/3*t^9 + 0(t^10) ? [f,g] = ellformaldifferential(E,8,'t); ? L' - f %3 = 0(t^8)

ellformalpoint(serprec, n)

If E is an elliptic curve, return the coordinates x(t), y(t) in the formal group of the elliptic curve E in the formal parameter t = -x/y at oo:

$$x = t^{-2} - a_1 t^{-1} - a_2 - a_3 t + \dots$$
$$y = -t^{-3} - a_1 t^{-2} - a_2 t^{-1} - a_3 + \dots$$

Return *n* terms (seriesprecision by default) of these two power series, whose coefficients are in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6]$.

? E = ellinit([0,0,1,-1,0]); [x,y] = ellformalpoint(E,8,'t); ? x %2 = t^-2 - t + t^2 - t^4 + 2*t^5 + 0(t^6) ? y %3 = -t^-3 + 1 - t + t^3 - 2*t^4 + 0(t^5) ? E = ellinit([0,1/2]); ellformalpoint(E,7) %4 = [x^-2 - 1/2*x^4 + 0(x^5), -x^-3 + 1/2*x^3 + 0(x^4)]

ellformalw(serprec, n)

Return the formal power series w attached to the elliptic curve E, in the variable t:

$$w(t) = t^{3}(1 + a_{1}t + (a_{2} + a_{1}^{2})t^{2} + \dots + O(t^{n})),$$

which is the formal expansion of -1/y in the formal parameter t := -x/y at *oo* (take n = series precision if n is omitted). The coefficients of w belong to $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6]$.

```
? E=ellinit([3,2,-4,-2,5]); ellformalw(E, 5, 't)
%1 = t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 0(t^8)
```

ellfromeqn()

Given a genus 1 plane curve, defined by the affine equation f(x, y) = 0, return the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of a Weierstrass equation for its Jacobian. This allows to recover a Weierstrass model for an elliptic curve given by a general plane cubic or by a binary quartic or biquadratic model. The function implements the $f : -- > f^*$ formulae of Artin, Tate and Villegas (Advances in Math. 198 (2005), pp. 366–382).

In the example below, the function is used to convert between twisted Edwards coordinates and Weierstrass coordinates.

? e = ellfromeqn(a*x^2+y^2 - (1+d*x^2*y^2)) %1 = [0, -a - d, 0, -4*d*a, 4*d*a^2 + 4*d^2*a] ? E = ellinit(ellfromeqn(y^2-x^2 - 1 +(121665/121666*x^2*y^2)),2^255-19); ? isprime(ellcard(E) / 8) %3 = 1

The elliptic curve attached to the sum of two cubes is given by

? ellfromeqn(x^3+y^3 - a) %1 = [0, 0, -9*a, 0, -27*a^2]

Congruent number problem. Let n be an integer, if $a^2 + b^2 = c^2$ and ab = 2n, then by substituting b by 2n/a in the first equation, we get $((a^2 + (2n/a)^2) - c^2)a^2 = 0$. We set x = a, y = ac.

```
? En = ellfromeqn((x^2 + (2*n/x)^2 - (y/x)^2)*x^2)
%1 = [0, 0, 0, -16*n^2, 0]
```

For example 23 is congruent since the curve has a point of infinite order, namely:

```
? ellheegner( ellinit(subst(En, n, 23)) )
%2 = [168100/289, 68053440/4913]
```

ellfromj()

Returns the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of a fixed elliptic curve with *j*-invariant *j*.

ellgenerators()

If E is an elliptic curve over the rationals, return a \mathbb{Z} -basis of the free part of the Mordell-Weil group attached to E. This relies on the elldata database being installed and referencing the curve, and so is only available for curves over \mathbb{Z} of small conductors. If E is an elliptic curve over a finite field \mathbb{F}_q as output by ellinit, return a minimal set of generators for the group $E(\mathbb{F}_q)$.

Caution. When the group is not cyclic, of shape $\mathbb{Z}/d_1\mathbb{Z}x\mathbb{Z}/d_2\mathbb{Z}$ with $d_2||d_1$, the points [P,Q] returned by eligenerators need not have order d_1 and d_2 : it is true that P has order d_1 , but we only know that Q is a generator of $E(\mathbb{F}_q)/\langle P \rangle$ and that the Weil pairing w(P,Q) has order d_2 , see ??ellgroup. If you need generators [P,R] with R of order d_2 , find x such that R = Q - [x]P has order d_2 by solving the discrete logarithm problem $[d_2]Q = [x]([d_2]P)$ in a cyclic group of order d_1/d_2 . This will be very expensive if d_1/d_2 has a large prime factor.

ellglobalred()

Let *E* be an ell structure as output by ellinit attached to an elliptic curve defined over a number field. This function calculates the arithmetic conductor and the global Tamagawa number *c*. The result [N, v, c, F, L] is slightly different if *E* is defined over \mathbb{Q} (domain D = 1 in ellinit) or over a number field (domain *D* is a number field structure, including nfinit(x) representing \mathbb{Q} !):

- N is the arithmetic conductor of the curve,
- v is an obsolete field, left in place for backward compatibility. If E is defined over Q, v gives the coordinate change for E to the standard minimal integral model (ellminimalmodel provides it in a cheaper way); if E is defined over another number field, v gives a coordinate change to an integral model (ellintegralmodel provides it in a cheaper way).
- c is the product of the local Tamagawa numbers c_p , a quantity which enters in the Birch and Swinnerton-Dyer conjecture,
- F is the factorization of N,
- L is a vector, whose *i*-th entry contains the local data at the *i*-th prime ideal divisor of N, i.e. L[i] = elllocalred(E, F[i, 1]). If E is defined over \mathbb{Q} , the local coordinate change has been deleted and replaced by a 0; if E is defined over another number field the local coordinate change to a local minimal model is given relative to the integral model afforded by v (so either start from an integral model so that v be trivial, or apply v first).

ellgroup(p, flag)

Let E be an ell structure as output by ellinit, attached to an elliptic curve E/K. We first describe the function when the field $K = \mathbb{F}_q$ is finite, it computes the structure of the finite abelian group $E(\mathbb{F}_q)$:

- if flag = 0, return the structure [] (trivial group) or [d₁] (nontrivial cyclic group) or [d₁, d₂] (noncyclic group) of E(𝔽_q) ℤ/d₁ℤxℤ/d₂ℤ, with d₂||d₁.
- if flag = 1, return a triple [h, cyc, gen], where h is the curve cardinality, cyc gives the group structure as a product of cyclic groups (as per flag = 0). More precisely, if $d_2 > 1$, the output is $[d_1d_2, [d_1, d_2], [P, Q]]$ where P is of order d_1 and [P, Q] generates the curve. **Caution**. It is not guaranteed that Q has order d_2 , which in the worst case requires an expensive discrete log computation. Only that ellweilpairing (E, P, Q, d_1) has order d_2 .

For other fields of definition and p defining a finite residue field \mathbb{F}_q , return the structure of the reduction of E: the argument p is best left omitted if $K = \mathbb{Q}_\ell$ (else we must have $p = \ell$) and must be a prime number ($K = \mathbb{Q}$) or prime ideal (K a general number field) with residue field \mathbb{F}_q otherwise. The curve is allowed to have bad reduction at p and in this case we consider the (cyclic) group of nonsingular points for the reduction of a minimal model at p.

If flag = 0, the equation not be minimal or even integral at p; of course, a minimal model will be more efficient.

If flag = 1, the requested generators depend on the model, which must then be minimal at p, otherwise an exception is thrown. Use ellintegralmodel and/or ellocalred first to reduce to this case.

```
? E = ellinit([0,1]); \\ y^2 = x^3 + 0.x + 1, defined over Q
? ellgroup(E, 7)
%2 = [6, 2] \\ Z/6 x Z/2, noncyclic
? E = ellinit([0,1] * Mod(1,11)); \\ defined over F_11
? ellgroup(E) \\ no need to repeat 11
%4 = [12]
? ellgroup(E, 11) \\ ... but it also works
%5 = [12]
? ellgroup(E, 13) \\ ouch, inconsistent input!
*** at top-level: ellgroup(E,13)
```

```
*** ^_____
*** ellgroup: inconsistent moduli in Rg_to_Fp:
11
13
? ellgroup(E, 7, 1)
%6 = [12, [6, 2], [[Mod(2, 7), Mod(4, 7)], [Mod(4, 7), Mod(4, 7)]]]
```

Let us now consider curves of bad reduction, in this case we return the structure of the (cyclic) group of nonsingular points, satisfying $\#E_{ns}(\mathbb{F}_p) = p - a_p$:

```
? E = ellinit([0,5]);
? ellgroup(E, 5, 1)
%2 = [5, [5], [[Mod(4, 5), Mod(2, 5)]]]
? ellap(E, 5)
%3 = 0 \\ additive reduction at 5
? E = ellinit([0,-1,0,35,0]);
? ellgroup(E, 5, 1)
%5 = [4, [4], [[Mod(2, 5), Mod(2, 5)]]]
? ellap(E, 5)
%6 = 1 \\ split multiplicative reduction at 5
? ellgroup(E, 7, 1)
%7 = [8, [8], [[Mod(3, 7), Mod(5, 7)]]]
? ellap(E, 7)
%8 = -1 \\ nonsplit multiplicative reduction at 7
```

ellheegner()

Let E be an elliptic curve over the rationals, assumed to be of (analytic) rank 1. This returns a nontorsion rational point on the curve, whose canonical height is equal to the product of the elliptic regulator by the analytic Sha.

This uses the Heegner point method, described in Cohen GTM 239; the complexity is proportional to the product of the square root of the conductor and the height of the point (thus, it is preferable to apply it to strong Weil curves).

ellheight(P, Q, precision)

Let *E* be an elliptic curve defined over $K = \mathbb{Q}$ or a number field, as output by ellinit; it needs not be given by a minimal model although the computation will be faster if it is.

- Without arguments P, Q, returns the Faltings height of the curve E using Deligne normalization. For a rational curve, the normalization is such that the function returns -(1/2)*log(ellminimalmodel(E), area).
- If the argument $P \in E(K)$ is present, returns the global Néron-Tate height h(P) of the point, using the normalization in Cremona's Algorithms for modular elliptic curves.
- If the argument $Q \in E(K)$ is also present, computes the value of the bilinear form (h(P+Q)-h(P-Q))/4.

ellheightmatrix(x, precision)

x being a vector of points, this function outputs the Gram matrix of x with respect to the Néron-Tate height, in other words, the (i, j) component of the matrix is equal to ellbil(:math:`E,x[i],x[j])`. The rank of this matrix, at least in some approximate sense, gives the rank of the set of points, and if x is a basis of the Mordell-Weil group of E, its determinant is equal to the regulator of E. Note our height normalization follows Cremona's Algorithms for modular elliptic curves: this matrix should be divided by 2 to be in accordance with, e.g., Silverman's normalizations.

ellidentify()

Look up the elliptic curve E, defined by an arbitrary model over \mathbb{Q} , in the elldata database. Return [[N, M, G], C] where N is the curve name in Cremona's elliptic curve database, M is the minimal model, G is a \mathbb{Z} -basis of the free part of the Mordell-Weil group $E(\mathbb{Q})$ and C is the change of coordinates from E to M, suitable for ellchangecurve.

ellinit(D, precision)

Initialize an ell structure, attached to the elliptic curve E. E is either

• a 5-component vector $[a_1, a_2, a_3, a_4, a_6]$ defining the elliptic curve with Weierstrass equation

 $Y^2 + a_1 XY + a_3 Y = X^3 + a_2 X^2 + a_4 X + a_6,$

• a 2-component vector $[a_4, a_6]$ defining the elliptic curve with short Weierstrass equation

$$Y^2 = X^3 + a_4 X + a_6,$$

• a character string in Cremona's notation, e.g. "11a1", in which case the curve is retrieved from the elldata database if available.

The optional argument D describes the domain over which the curve is defined:

- the t_INT 1 (default): the field of rational numbers Q.
- a t_INT p, where p is a prime number: the prime finite field \mathbb{F}_p .
- an t_INTMOD Mod(a, p), where p is a prime number: the prime finite field \mathbb{F}_p .
- a t_FFELT, as returned by ffgen: the corresponding finite field \mathbb{F}_q .
- a t_PADIC, O(pⁿ): the field Q_p, where p-adic quantities will be computed to a relative accuracy of n digits. We advise to input a model defined over Q for such curves. In any case, if you input an approximate model with t_PADIC coefficients, it will be replaced by a lift to Q (an exact model "close" to the one that was input) and all quantities will then be computed in terms of this lifted model, at the given accuracy.
- a t_REAL x: the field \mathbb{C} of complex numbers, where floating point quantities are by default computed to a relative accuracy of precision(x). If no such argument is given, the value of realprecision at the time ellinit is called will be used.
- a number field K, given by a nf or bnf structure; a bnf is required for ellminimalmodel.
- a prime ideal p, given by a prid structure; valid if x is a curve defined over a number field K and the equation is integral and minimal at p.

This argument D is indicative: the curve coefficients are checked for compatibility, possibly changing D; for instance if D = 1 and an t_INTMOD is found. If inconsistencies are detected, an error is raised:

If the curve coefficients are too general to fit any of the above domain categories, only basic operations, such as point addition, will be supported later.

If the curve (seen over the domain D) is singular, fail and return an empty vector [].

? E = ellinit([0,0,0,0,1]); \\ y^2 = x^3 + 1, over Q ? E = ellinit([0,1]); \\ the same curve, short form ? E = ellinit("36a1"); \\ sill the same curve, Cremona's notations ? E = ellinit([0,1], 2) \\ over F2: singular curve %4 = [] ? E = ellinit(['a4,'a6] * Mod(1,5)); \\ over F_5[a4,a6], basic support !

The result of ellinit is an *ell* structure. It contains at least the following information in its components:

 $a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j.$

All are accessible via member functions. In particular, the discriminant is :math: `E.disc`, and the j-invariant is :math: `E.j`.

```
? E = ellinit([a4, a6]);
? E.disc
%2 = -64*a4^3 - 432*a6^2
? E.j
%3 = -6912*a4^3/(-4*a4^3 - 27*a6^2)
```

Further components contain domain-specific data, which are in general dynamic: only computed when needed, and then cached in the structure.

```
? E = ellinit([2,3], 10^60+7); \\ E over F_p, p large
? ellap(E)
time = 4,440 ms.
%2 = -1376268269510579884904540406082
? ellcard(E); \\ now instantaneous !
time = 0 ms.
? ellgenerators(E);
time = 5,965 ms.
? ellgenerators(E); \\ second time instantaneous
time = 0 ms.
```

See the description of member functions related to elliptic curves at the beginning of this section.

ellintegralmodel(v)

Let E be an ell structure over a number field K or \mathbb{Q}_p . This function returns an integral model. If v is present, sets v = [u, 0, 0, 0] to the corresponding change of variable: the return value is identical to that of ellchangecurve(E, v).

? e = ellinit([1/17,1/42]); ? e = ellintegralmodel(e,&v); ? e[1..5] %3 = [0, 0, 0, 15287762448, 3154568630095008] ? v %4 = [1/714, 0, 0, 0]

ellisdivisible(P, n, Q)

Given E/K a number field and P in E(K) return 1 if P = [n]R for some R in E(K) and set Q to one such R; and return 0 otherwise. The integer $n \ge 0$ may be given as ellxn(E,n), if many points need to be tested.

```
? K = nfinit(polcyclo(11,t));
? E = ellinit([0,-1,1,0,0], K);
? P = [0,0];
? ellorder(E,P)
%4 = 5
? ellisdivisible(E,P,5, &Q)
%5 = 1
? lift(Q)
%6 = [-t^7-t^6-t^5-t^4+1, -t^9-2*t^8-2*t^7-3*t^6-3*t^5-2*t^4-2*t^3-t^2-1]
? ellorder(E, Q)
%7 = 25
```

The algebraic complexity of the underlying algorithm is in $O(n^4)$, so it is advisable to first factor n, then use a chain of checks attached to the prime divisors of n: the function will do it itself unless n is given in ellxn form.

ellisogeny(G, only_image, x, y)

Given an elliptic curve E, a finite subgroup G of E is given either as a generating point P (for a cyclic G) or as a polynomial whose roots vanish on the x-coordinates of the nonzero elements of G (general case and more efficient if available). This function returns the $[a_1, a_2, a_3, a_4, a_6]$ invariants of the quotient elliptic curve E/Gand (if *only_image* is zero (the default)) a vector of rational functions [f, g, h] such that the isogeny $E \to E/G$ is given by $(x, y) : -- > (f(x)/h(x)^2, g(x, y)/h(x)^3)$.

```
? E = ellinit([0,1]);
? elltors(E)
%2 = [6, [6], [[2, 3]]]
? ellisogeny(E, [2,3], 1) \\ Weierstrass model for E/<P>
%3 = [0, 0, 0, -135, -594]
? ellisogeny(E,[-1,0])
%4 = [[0,0,0,-15,22], [x^3+2*x^2+4*x+3, y*x^3+3*y*x^2-2*y, x+1]]
```

ellisogenyapply(g)

Given an isogeny of elliptic curves $f: E' \to E$ (being the result of a call to ellisogeny), apply f to g:

- if g is a point P in the domain of f, return the image f(P);
- if $g: E^{"} \to E'$ is a compatible isogeny, return the composite isogeny $fog: E^{"} \to E$.

```
? one = ffgen(101, 't)^0;
? E = ellinit([6, 53, 85, 32, 34] * one);
? P = [84, 71] * one;
? ellorder(E, P)
%4 = 5
? [F, f] = ellisogeny(E, P); \\ f: E->F = E/<P>
? ellisogenyapply(f, P)
%6 = [0]
? F = ellinit(F);
? Q = [89, 44] * one;
? ellorder(F, Q)
%9 = 2
? [G, g] = ellisogeny(F, Q); \\ g: F->G = F/<Q>
? gof = ellisogenyapply(g, f); \\ gof: E -> G
```

ellisomat(p,fl)

Given an elliptic curve E defined over a number field K, compute representatives of the isomorphism classes of elliptic curves defined over K and K-isogenous to E. We assume that E does not have CM over K (otherwise

that set would be infinite). For any such curve E_i , let $f_i : E \to E_i$ be a rational isogeny of minimal degree and let $g_i : E_i \to E$ be the dual isogeny; and let M be the matrix such that $M_{i,j}$ is the minimal degree for an isogeny $E_i \to E_j$.

The function returns a vector [L, M] where L is a list of triples $[E_i, f_i, g_i]$ (flag = 0), or simply the list of E_i (flag = 1, which saves time). The curves E_i are given in $[a_4, a_6]$ form and the first curve E_1 is isomorphic to E by f_1 .

If p is set, it must be a prime number; in this which case only isogenies of degree a power of p are considered.

Over a number field, the possible isogeny degrees are determined by Billerey algorithm.

```
? E = ellinit("14a1");
? [L,M] = ellisomat(E);
? LE = apply(x->x[1], L) \setminus list of curves
%3 = [[215/48,-5291/864],[-675/16,6831/32],[-8185/48,-742643/864],
[-1705/48,-57707/864],[-13635/16,306207/32],[-131065/48,-47449331/864]]
? L[2][2] \\ isogeny f_2
\%4 = [x^3+3/4*x^2+19/2*x-311/12],
1/2*x^{4}+(y+1)*x^{3}+(y-4)*x^{2}+(-9*y+23)*x+(55*y+55/2),x+1/3
? L[2][3] \setminus dual isogeny g_2
\%5 = [1/9*x^{3}-1/4*x^{2}-141/16*x+5613/64],
-1/18*x^{4}+(1/27*y-1/3)*x^{3}+(-1/12*y+87/16)*x^{2}+(49/16*y-48)*x
+(-3601/64*y+16947/512), x-3/4]
? apply(E->ellidentify(ellinit(E))[1][1], LE)
%6 = ["14a1","14a4","14a3","14a2","14a6","14a5"]
? M
%7 =
[1 3 3 2 6 6]
[3 1 9 6 2 18]
[3 9 1 6 18 2]
[2 6 6 1 3 3]
[6 2 18 3 1 9]
[6 18 2 3 9 1]
```

ellisoncurve(z)

Gives 1 (i.e. true) if the point z is on the elliptic curve E, 0 otherwise. If E or z have imprecise coefficients, an attempt is made to take this into account, i.e. an imprecise equality is checked, not a precise one. It is allowed for z to be a vector of points in which case a vector (of the same type) is returned.

ellisotree()

Given an elliptic curve E defined over \mathbb{Q} or a set of \mathbb{Q} -isogenous curves as given by ellisomat, return a pair [L, M] where

- L lists the minimal models of the isomorphism classes of elliptic curves Q-isogenous to E (or in the set of isogenous curves),
- M is the adjacency matrix of the prime degree isogenies tree: there is an edge from E_i to E_j if there is an isogeny E_i → E_j of prime degree such that the Néron differential forms are preserved.

```
? E = ellinit("14a1");
? [L,M] = ellisotree(E);
? M
%3 =
[0 0 3 2 0 0]
[3 0 0 0 2 0]
[0 0 0 0 0 2]
[0 0 0 0 0 3]
[0 0 0 3 0 0]
[0 0 0 0 0 0]
? [L2,M2] = ellisotree(ellisomat(E,2,1));
%4 =
[0 2]
[0 0]
? [L3,M3] = ellisotree(ellisomat(E,3,1));
? M3
%6 =
[0 0 3]
[3 0 0]
[0 0 0]
```

Compare with the result of ellisomat.

? [L,M]=ellisomat(E,,1); ? M %7 = [1 3 3 2 6 6] [3 1 9 6 2 18] [3 9 1 6 18 2] [2 6 6 1 3 3] [6 2 18 3 1 9] [6 18 2 3 9 1]

ellissupersingular(p)

Return 1 if the elliptic curve E defined over a number field, \mathbb{Q}_p or a finite field is supersingular at p, and 0 otherwise. If the curve is defined over a number field, p must be explicitly given, and must be a prime number, resp. a maximal ideal, if the curve is defined over \mathbb{Q} , resp. a general number field: we return 1 if and only if E has supersingular good reduction at p.

Alternatively, E can be given by its j-invariant in a finite field. In this case p must be omitted.

```
? setrand(1); \\ make the choice of g deterministic
? g = ffprimroot(ffgen(7^5))
%1 = 4*x^4 + 5*x^3 + 6*x^2 + 5*x + 6
? [g^n | n <- [1 .. 7^5 - 1], ellissupersingular(g^n)]
%2 = [6]
? K = nfinit(y^3-2); P = idealprimedec(K, 2)[1];
? E = ellinit([y,1], K);
? ellissupersingular(E, P)
%5 = 1
? Q = idealprimedec(K,5)[1];
? ellissupersingular(E, Q)
%6 = 0
```

ellj(precision)

Elliptic j-invariant. x must be a complex number with positive imaginary part, or convertible into a power series or a p-adic number with positive valuation.

elllocalred(p)

Calculates the Kodaira type of the local fiber of the elliptic curve E at p. E must be an ell structure as output by ellinit, over \mathbb{Q}_{ℓ} (p better left omitted, else equal to ℓ) over \mathbb{Q} (p a rational prime) or a number field K(p a maximal ideal given by a prid structure). The result is a 4-component vector [f, kod, v, c]. Here f is the exponent of p in the arithmetic conductor of E, and kod is the Kodaira type which is coded as follows:

1 means good reduction (type I:math:_0), 2, 3 and 4 mean types II, III and IV respectively, $4+\nu$ with $\nu > 0$ means type I:math:_nu; finally the opposite values -1, -2, etc. refer to the starred types I:math:_ 0^* , II:math:^*, etc. The third component v is itself a vector [u, r, s, t] giving the coordinate changes done during the local reduction; u = 1 if and only if the given equation was already minimal at p. Finally, the last component c is the local Tamagawa number c_p .

elllog(P, G, o)

Given two points P and G on the elliptic curve E/\mathbb{F}_q , returns the discrete logarithm of P in base G, i.e. the smallest nonnegative integer n such that P = [n]G. See znlog for the limitations of the underlying discrete log algorithms. If present, o represents the order of G, see DLfun (in the PARI manual); the preferred format for this parameter is [N, factor(N)], where N is the order of G.

If no o is given, assume that G generates the curve. The function also assumes that P is a multiple of G.

```
? a = ffgen(ffinit(2,8),'a);
? E = ellinit([a,1,0,0,1]); \\ over F_{2^8}
? x = a^3; y = ellordinate(E,x)[1];
? P = [x,y]; G = ellmul(E, P, 113);
? ord = [242, factor(242)]; \\ P generates a group of order 242. Initialize.
? ellorder(E, G, ord)
%4 = 242
? e = elllog(E, P, G, ord)
%5 = 15
? ellmul(E,G,e) == P
%6 = 1
```

elllseries(s, A, precision)

This function is deprecated, use lfun(E,s) instead.

E being an elliptic curve, given by an arbitrary model over \mathbb{Q} as output by ellipsit, this function computes the value of the *L*-series of *E* at the (complex) point *s*. This function uses an $O(N^{1/2})$ algorithm, where *N* is the

conductor.

The optional parameter A fixes a cutoff point for the integral and is best left omitted; the result must be independent of A, up to realprecision, so this allows to check the function's accuracy.

ellminimaldisc()

E being an elliptic curve defined over a number field output by ellinit, return the minimal discriminant ideal of E.

ellminimalmodel(v)

Let *E* be an ell structure over a number field *K*. This function determines whether *E* admits a global minimal integral model. If so, it returns it and sets v = [u, r, s, t] to the corresponding change of variable: the return value is identical to that of ellchangecurve(E, v).

Else return the (nonprincipal) Weierstrass class of E, i.e. the class of $\prod p^{(v_p \Delta - \delta_p)/12}$ where $\Delta = E.disc$ is the model's discriminant and p_p^{δ} is the local minimal discriminant. This function requires either that E be defined over the rational field \mathbb{Q} (with domain D = 1 in ellinit), in which case a global minimal model always exists, or over a number field given by a *bnf* structure. The Weierstrass class is given in bnfisprincipal format, i.e. in terms of the K.gen generators.

The resulting model has integral coefficients and is everywhere minimal, the coefficients a_1 and a_3 are reduced modulo 2 (in terms of the fixed integral basis K.zk) and a_2 is reduced modulo 3. Over \mathbb{Q} , we further require that a_1 and a_3 be 0 or 1, that a_2 be 0 or 1 and that u > 0 in the change of variable: both the model and the change of variable v are then unique.

```
? e = ellinit([6,6,12,55,233]); \\ over Q
? E = ellminimalmodel(e, &v);
? E[1..5]
%3 = [0, 0, 0, 1, 1]
? v
%4 = [2, -5, -3, 9]
```

```
? K = bnfinit(a^2-65); \\ over a nonprincipal number field
? K.cyc
%2 = [2]
? u = Mod(8+a, K.pol);
? E = ellinit([1,40*u+1,0,25*u^2,0], K);
? ellminimalmodel(E) \\ no global minimal model exists over Z_K
%6 = [1]~
```

ellminimaltwist(flag)

Let *E* be an elliptic curve defined over \mathbb{Q} , return a discriminant *D* such that the twist of *E* by *D* is minimal among all possible quadratic twists, i.e. if flag = 0, its minimal model has minimal discriminant, or if flag = 1, it has minimal conductor.

In the example below, we find a curve with j-invariant 3 and minimal conductor.

```
? E = ellminimalmodel(ellinit(ellfromj(3)));
? ellglobalred(E)[1]
%2 = 357075
? D = ellminimaltwist(E,1)
%3 = -15
? E2 = ellminimalmodel(ellinit(elltwist(E,D)));
? ellglobalred(E2)[1]
%5 = 14283
```

In the example below, flag = 0 and flag = 1 give different results.

```
? E = ellinit([1,0]);
? D0 = ellminimaltwist(E,0)
%7 = 1
? D1 = ellminimaltwist(E,1)
%8 = 8
? E0 = ellminimalmodel(ellinit(elltwist(E,D0)));
? [E0.disc, ellglobalred(E0)[1]]
%10 = [-64, 64]
? E1 = ellminimalmodel(ellinit(elltwist(E,D1)));
? [E1.disc, ellglobalred(E1)[1]]
%12 = [-4096, 32]
```

ellmoddegree()

e being an elliptic curve defined over \mathbb{Q} output by ellinit, compute the modular degree of *e* divided by the square of the Manin constant *c*. It is conjectured that c = 1 for the strong Weil curve in the isogeny class (optimal quotient of $J_0(N)$) and this can be proven using ellweilcurve when the conductor *N* is moderate.

```
? E = ellinit("11a1"); \\ from Cremona table: strong Weil curve and c = 1
? [v,smith] = ellweilcurve(E); smith \\ proof of the above
%2 = [[1, 1], [5, 1], [1, 1/5]]
? ellmoddegree(E)
%3 = 1
? [ellidentify(e)[1][1] | e<-v]
%4 = ["11a1", "11a2", "11a3"]
? ellmoddegree(ellinit("11a3"))
%5 = 5
? ellmoddegree(ellinit("11a3"))
%6 = 1/5
```

The modular degree of 11a1 is 1 (because ellweilcurve or Cremona's table prove that the Manin constant is 1 for this curve); the output of ellweilcurve also proves that the Manin constants of 11a2 and 11a3 are 1 and 5 respectively, so the actual modular degree of both 11a2 and 11a3 is 5.

ellmul(z, n)

Computes [n]z, where z is a point on the elliptic curve E. The exponent n is in \mathbb{Z} , or may be a complex quadratic integer if the curve E has complex multiplication by n (if not, an error message is issued).

```
? Ei = ellinit([1,0]); z = [0,0];
? ellmul(Ei, z, 10)
%2 = [0] \\ unsurprising: z has order 2
? ellmul(Ei, z, I)
%3 = [0, 0] \\ Ei has complex multiplication by Z[i]
? ellmul(Ei, z, quadgen(-4))
%4 = [0, 0] \\ an alternative syntax for the same query
? Ej = ellinit([0,1]); z = [-1,0];
? ellmul(Ej, z, I)
*** at top-level: ellmul(Ej,z,I)
*** at top-level: ellmul(Ej,z,I)
*** ellmul: not a complex multiplication in ellmul.
? ellmul(Ej, z, 1+quadgen(-3))
%6 = [1 - w, 0]
```

The simple-minded algorithm for the CM case assumes that we are in characteristic 0, and that the quadratic order to which n belongs has small discriminant.

ellneg(z)

Opposite of the point z on elliptic curve E.

ellnonsingularmultiple(P)

Given an elliptic curve E/\mathbb{Q} (more precisely, a model defined over \mathbb{Q} of a curve) and a rational point $P \in E(\mathbb{Q})$, returns the pair [R, n], where n is the least positive integer such that R := [n]P has good reduction at every prime. More precisely, its image in a minimal model is everywhere nonsingular.

? e = ellinit("57a1"); P = [2,-2]; ? ellnonsingularmultiple(e, P) %2 = [[1, -1], 2] ? e = ellinit("396b2"); P = [35, -198]; ? [R,n] = ellnonsingularmultiple(e, P); ? n %5 = 12

ellorder(z, o)

Gives the order of the point z on the elliptic curve E, defined over a finite field or a number field. Return (the impossible value) zero if the point has infinite order.

```
? E = ellinit([-157^2, 0]); \setminus the "157-is-congruent" curve
? P = [2,2]; ellorder(E, P)
\%2 = 2
? P = ellheegner(E); ellorder(E, P) \\ infinite order
\%3 = 0
? K = nfinit(polcyclo(11,t)); E=ellinit("11a3", K); T = elltors(E);
? ellorder(E, T.gen[1])
\%5 = 25
? E = ellinit(ellfromj(ffgen(5^10)));
? ellcard(E)
%7 = 9762580
? P = random(E); ellorder(E, P)
%8 = 4881290
? p = 2^160+7; E = ellinit([1,2], p);
? N = ellcard(E)
\%9 = 1461501637330902918203686560289225285992592471152
? o = [N, factor(N)];
? for(i=1,100, ellorder(E,random(E)))
time = 260 \text{ ms}.
```

The parameter o, is now mostly useless, and kept for backward compatibility. If present, it represents a nonzero multiple of the order of z, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the cardinality of the curve. It is no longer needed since PARI is now able to compute it over large finite fields (was restricted to small prime fields at the time this feature was introduced), *and* caches the result in E so that it is computed and factored only once. Modifying the last example, we see that including this extra parameter provides no improvement:

```
? o = [N, factor(N)];
? for(i=1,100, ellorder(E,random(E),o))
time = 260 ms.
```

ellordinate(x, precision)

Gives a 0, 1 or 2-component vector containing the y-coordinates of the points of the curve E having x as x-coordinate.

ellpadicL(*p*, *n*, *s*, *r*, *D*)

Returns the value (or r-th derivative) on a character χ^s of \mathbb{Z}_p^* of the p-adic L-function of the elliptic curve E/\mathbb{Q} , twisted by D, given modulo p^n .

Characters. The set of continuous characters of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$ is identified to \mathbb{Z}_p^* via the cyclotomic character χ with values in $\overline{\mathbb{Q}_p}^*$. Denote by $\tau : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ the Teichmüller character, with values in the (p-1)-th roots of 1 for p! = 2, and -1, 1 for p = 2; finally, let $\langle \chi \rangle \geq \chi \tau^{-1}$, with values in $1 + 2p\mathbb{Z}_p$. In GP, the continuous character of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$ given by $\langle \chi \rangle^{s_1} \tau^{s_2}$ is represented by the pair of integers $s = (s_1, s_2)$, with $s_1 \in \mathbb{Z}_p$ and $s_2 modp - 1$ for p > 2, (resp. mod 2 for p = 2); s may be also an integer, representing (s, s) or χ^s .

The :math:`p-adic L function.` The p-adic L function L_p is defined on the set of continuous characters of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$, as $\int_{\mathbb{Z}_{p}^{*}} \chi^{s} d\mu$ for a certain p-adic distribution μ on \mathbb{Z}_{p}^{*} . The derivative is given by

$$L_p^{(r)}(E,\chi^s) = \int_{\mathbb{Z}_p^*} \log_p^r(a) \chi^s(a) d\mu(a)$$

More precisely:

• When E has good supersingular reduction, L_p takes its values in $D := H^1_{dR}(E/\mathbb{Q}) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ and satisfies

where: math: `F' is the Frobenius,: math: `L(E,1)' is the value of the complex: math: `L' functionat: math: `1',: math' is the value of the complex: math' is the value of the complex is the value of the complex is the value of the complex is the value of the valu

The function returns the components of $L_p^{(r)}(E, \chi^s)$ in the basis $(\omega, F\omega)$.

• When E has ordinary good reduction, this method only defines the projection of $L_p(E, \chi^s)$ on the α eigenspace, where α is the unit eigenvalue for F. This is what the function returns. We have

$$(1 - \alpha^{-1})^{-2} L_{p,\alpha}(E, \chi^0) = L(E, 1) / \Omega.$$

Two supersingular examples:

```
? cxL(e) = bestappr( ellL1(e) / e.omega[1] );
? e = ellinit("17a1"); p=3; \\ supersingular, a3 = 0
? L = ellpadicL(e,p,4);
? F = [0,-p;1,ellap(e,p)]; \\ Frobenius matrix in the basis (omega,F(omega))
? (1-p^(-1)*F)^-2 * L / cxL(e)
%5 = [1 + 0(3^5), 0(3^5)]~ \\ [1,0]~
? e = ellinit("116a1"); p=3; \\ supersingular, a3 != 0~
? L = ellpadicL(e,p,4);
? F = [0,-p; 1,ellap(e,p)];
? (1-p^(-1)*F)^-2*L~ / cxL(e)
%9 = [1 + 0(3^4), 0(3^5)]~
```

Good ordinary reduction:

? e = ellinit("17a1"); p=5; ap = ellap(e,p) %1 = -2 \\ ordinary ? L = ellpadicL(e,p,4) %2 = 4 + 3*5 + 4*5^2 + 2*5^3 + 0(5^4) ? al = padicappr(x^2 - ap*x + p, ap + 0(p^7))[1]; ? (1-al^(-1))^(-2) * L / cxL(e) %4 = 1 + 0(5^4) Twist and Teichmüller:

? e = ellinit("17a1"); p=5; \\ ordinary \\ 2nd derivative at tau^1, twist by -7 ? ellpadicL(e, p, 4, [0,1], 2, -7) %2 = 2*5^2 + 5^3 + 0(5^4)

We give an example of non split multiplicative reduction (see ellpadicbsd for more examples).

? e=ellinit("15a1"); p=3; n=5; ? L = ellpadicL(e,p,n) %2 = 2 + 3 + 3^2 + 3^3 + 3^4 + 0(3^5) ? (1 - ellap(e,p))^(-1) * L / cxL(e) %3 = 1 + 0(3^5)

This function is a special case of mspadicL and it also appears as the first term of mspadicseries:

```
? e = ellinit("17a1"); p=5;
? L = ellpadicL(e,p,4)
%2 = 4 + 3*5 + 4*5^2 + 2*5^3 + 0(5^4)
? [M,phi] = msfromell(e, 1);
? Mp = mspadicinit(M, p, 4);
? mu = mspadicmoments(Mp, phi);
? mspadicL(mu)
%6 = 4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 0(5^6)
? mspadicseries(mu)
%7 = (4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 0(5^6))
+ (3 + 3*5 + 5^2 + 5^3 + 0(5^4))*x
+ (2 + 3*5 + 5^2 + 0(5^3))*x^2
+ (3 + 4*5 + 4*5^2 + 0(5^3))*x^3
+ (3 + 2*5 + 0(5^2))*x^4 + 0(x^5)
```

These are more cumbersome than ellpadicL but allow to compute at different characters, or successive derivatives, or to twist by a quadratic character essentially for the cost of a single call to ellpadicL due to precomputations.

ellpadicbsd(p, n, D)

Given an elliptic curve E over \mathbb{Q} , its quadratic twist E_D and a prime number p, this function is a p-adic analog of the complex functions ellanalyticrank and ellbsd. It calls ellpadicL with initial accuracy p^n and may increase it internally; it returns a vector $[r, L_p]$ where

- L_p is a *p*-adic number (resp. a pair of *p*-adic numbers if *E* has good supersingular reduction) defined modulo p^N , conjecturally equal to R_pS , where R_p is the *p*-adic regulator as given by ellpadicregulator (in the basis $(\omega, F\omega)$) and *S* is the cardinal of the Tate-Shafarevich group for the quadratic twist E_D .
- r is an upper bound for the analytic rank of the p-adic L-function attached to E_D : we know for sure that the *i*-th derivative of $L_p(E_D, .)$ at χ^0 is $O(p^N)$ for all i < r and that its r-th derivative is nonzero; it is expected that the true analytic rank is equal to the rank of the Mordell-Weil group $E_D(\mathbb{Q})$, plus 1 if the reduction of E_D at p is split multiplicative; if r = 0, then both the analytic rank and the Mordell-Weil rank are unconditionnally 0.

Recall that the *p*-adic BSD conjecture (Mazur, Tate, Teitelbaum, Bernardi, Perrin-Riou) predicts an explicit link between R_pS and

$$(1-p^{-1}F)^{-2}.L_n^{(r)}(E_D,\chi^0)/r!$$

where r is the analytic rank of the p-adic L-function attached to E_D and F is the Frobenius on H_{dR}^1 ; see ellpadicL for definitions.

```
? E = ellinit("11a1"); p = 7; n = 5; \\ good ordinary
? ellpadicbsd(E, 7, 5) \setminus rank 0,
\%2 = [0, 1 + 0(7^5)]
? E = ellinit("91a1"); p = 7; n = 5; \setminus non split multiplicative
? [r,Lp] = ellpadicbsd(E, p, n)
\%5 = [1, 2*7 + 6*7^2 + 3*7^3 + 7^4 + 0(7^5)]
? R = ellpadicregulator(E, p, n, E.gen)
\%6 = 2*7 + 6*7^2 + 3*7^3 + 7^4 + 5*7^5 + 0(7^6)
? sha = Lp/R
\%7 = 1 + 0(7^{4})
? E = ellinit("91b1"); p = 7; n = 5; \\ split multiplicative
? [r,Lp] = ellpadicbsd(E, p, n)
\%9 = [2, 2*7 + 7^2 + 5*7^3 + 0(7^4)]
? ellpadicregulator(E, p, n, E.gen)
\%10 = 2*7 + 7^{2} + 5*7^{3} + 6*7^{4} + 2*7^{5} + 0(7^{6})
? [rC, LC] = ellanalyticrank(E);
? [r, rC]
12 = [2, 1] \setminus r = rC+1 because of split multiplicative reduction
? E = ellinit("53a1"); p = 5; n = 5; \\ supersingular
? [r, Lp] = ellpadicbsd(E, p, n);
? r
%15 = 1
? Lp
%16 = [3*5 + 2*5^2 + 2*5^5 + 0(5^6), \setminus
5 + 3*5^2 + 4*5^3 + 2*5^4 + 5^5 + 0(5^6)
? R = ellpadicregulator(E, p, n, E.gen)
%17 = [3*5 + 2*5^2 + 2*5^5 + 0(5^6), 5 + 3*5^2 + 4*5^3 + 2*5^4 + 0(5^5)]
\ expect Lp = R*#Sha, hence (conjecturally) #Sha = 1
? E = ellinit("84a1"); p = 11; n = 6; D = -443;
? [r,Lp] = ellpadicbsd(E, 11, 6, D) \\ Mordell-Weil rank 0, no regulator
\%19 = [0, 3 + 2*11 + 0(11^6)]
? lift(Lp) \setminus expected cardinal for Sha is 5^2
\%20 = 25
? ellpadicbsd(E, 3, 12, D) \\ at 3
\%21 = [1, 1 + 2*3 + 2*3*2 + 0(3*8)]
? ellpadicbsd(E, 7, 8, D) \setminus and at 7
\%22 = [0, 4 + 3*7 + 0(7*8)]
```

ellpadicfrobenius(p, n)

If p > 2 is a prime and E is an elliptic curve on \mathbb{Q} with good reduction at p, return the matrix of the Frobenius endomorphism φ on the crystalline module $D_p(E) = \mathbb{Q}_p \otimes H^1_{dR}(E/\mathbb{Q})$ with respect to the basis of the given model $(\omega, \eta = x\omega)$, where $\omega = dx/(2y + a_1x + a_3)$ is the invariant differential. The characteristic polynomial of φ is $x^2 - a_p x + p$. The matrix is computed to absolute p-adic precision p^n .

? E = ellinit([1,-1,1,0,0]); ? F = ellpadicfrobenius(E,5,3);

? lift(F)
%3 =
[120 29]
[55 5]
? charpoly(F)
%4 = x^2 + 0(5^3)*x + (5 + 0(5^3))
? ellap(E, 5)
%5 = 0

ellpadicheight(p, n, P, Q)

Cyclotomic *p*-adic height of the rational point *P* on the elliptic curve *E* (defined over \mathbb{Q}), given to *n p*-adic digits. If the argument *Q* is present, computes the value of the bilinear form (h(P+Q) - h(P-Q))/4.

Let $D := H_{dR}^1(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ be the \mathbb{Q}_p vector space spanned by ω (invariant differential $dx/(2y + a_1x + a_3)$ related to the given model) and $\eta = x\omega$. Then the cyclotomic *p*-adic height h_E associates to $P \in E(\mathbb{Q})$ an element $f\omega + g\eta$ in D. This routine returns the vector [f, g] to n *p*-adic digits. If $P \in E(\mathbb{Q})$ is in the kernel of reduction mod p and if its reduction at all finite places is non singular, then $g = -(\log_E P)^2$, where \log_E is the logarithm for the formal group of E at p.

If furthermore the model is of the form $Y^2 = X^3 + aX + b$ and P = (x, y), then

$$f = \log_{n}(denominator(x)) - 2\log_{n}(\sigma(P))$$

where $\sigma(P)$ is given by ellsigma(E, P).

Recall (Advanced topics in the arithmetic of elliptic curves, Theorem 3.2) that the local height function over the complex numbers is of the form

$$\lambda(z) = -\log(\|E.disc\|)/6 + \Re(z\eta(z)) - 2\log(\sigma(z)).$$

(N.B. our normalization for local and global heights is twice that of Silverman's).

```
? E = ellinit([1,-1,1,0,0]); P = [0,0];
? ellpadicheight(E,5,3, P)
%2 = [3*5 + 5^2 + 2*5^3 + 0(5^4), 5^2 + 4*5^4 + 0(5^5)]
? E = ellinit("11a1"); P = [5,5]; \\ torsion point
? ellpadicheight(E,19,6, P)
%4 = [0, 0]
? E = ellinit([0,0,1,-4,2]); P = [-2,1];
? ellpadicheight(E,3,3, P)
%6 = [2*3^2 + 2*3^3 + 3^4 + 0(3^5), 2*3^2 + 3^4 + 0(3^5)]
? ellpadicheight(E,3,5, P, elladd(E,P,P))
%7 = [3^2 + 2*3^3 + 0(3^7), 3^2 + 3^3 + 2*3^4 + 3^5 + 0(3^7)]
```

• When E has good ordinary reduction at p or non split multiplicative reduction, the "canonical" p-adic height is given by

```
s2 = ellpadics2(E,p,n);
ellpadicheight(E, p, n, P) * [1,-s2]~
```

Since s_2 does not depend on P, it is preferable to compute it only once:

```
? E = ellinit("5077a1"); p = 5; n = 7; \\ rank 3
? s2 = ellpadics2(E,p,n);
? M = ellpadicheightmatrix(E,p, n, E.gen) * [1,-s2]~;
? matdet(M) \\ p-adic regulator on the points in E.gen
%4 = 5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 0(5^7)
```

• When E has split multiplicative reduction at p (Tate curve), the "canonical" p-adic height is given by

```
\begin{split} Ep &= ellinit(E[1..5], 0(p^{(n)}); \ \ E \ seen \ as \ a \ Tate \ curve \ over \ Qp \\ [u2,u,q] &= Ep.tate; \\ ellpadicheight(E, p, n, P) \ \ [1,-s2 \ + \ 1/log(q)/u2]] \sim \end{split}
```

where s_2 is as above. For example,

```
? E = ellinit("91b1"); P =[-1, 3]; p = 7; n = 5;
? Ep = ellinit(E[1..5], 0(p^(n)));
? s2 = ellpadics2(E,p,n);
? [u2,u,q] = Ep.tate;
? H = ellpadicheight(E,p, n, P) * [1,-s2 + 1/log(q)/u2]~
%5 = 2*7 + 7^2 + 5*7^3 + 6*7^4 + 2*7^5 + 0(7^6)
```

These normalizations are chosen so that *p*-adic BSD conjectures are easy to state, see ellpadicbsd.

ellpadicheightmatrix(p, n, Q)

Q being a vector of points, this function returns the "Gram matrix" [F, G] of the cyclotomic *p*-adic height h_E with respect to the basis (ω, η) of $D = H^1_{dR}(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ given to *n p*-adic digits. In other words, if ellpadicheight(E, p, n, Q[i], Q[j]) = [f, g], corresponding to $f\omega + g\eta$ in *D*, then F[i, j] = f and G[i, j] = g.

```
? E = ellinit([0,0,1,-7,6]); Q = [[-2,3],[-1,3]]; p = 5; n = 5;
? [F,G] = ellpadicheightmatrix(E,p,n,Q);
? lift(F) \\ p-adic entries, integral approximation for readability
%3 =
[2364 3100]
[3100 3119]
? G
%4 =
[25225 46975]
[46975 61850]
? [F,G] * [1,-ellpadics2(E,p,n)]~
%5 =
[4 + 2*5 + 4*5^2 + 3*5^3 + 0(5^5) 4*5^2 + 4*5^3 + 5^4 + 0(5^5)]
[ 4*5^2 + 4*5^3 + 5^4 + 0(5^5) 4 + 3*5 + 4*5^2 + 4*5^3 + 5^4 + 0(5^5)]
```

ellpadiclambdamu(p, D, i)

Let p be a prime number and let E/\mathbb{Q} be a rational elliptic curve with good or bad multiplicative reduction at p. Return the Iwasawa invariants λ and μ for the p-adic L function $L_p(E)$, twisted by (D/.) and the *i*-th power of the Teichmüller character τ , see ellpadicL for details about $L_p(E)$. Let χ be the cyclotomic character and choose γ in $Gal(\mathbb{Q}_p(\mu_{p^oo})/\mathbb{Q}_p)$ such that $\chi(\gamma) = 1 + 2p$. Let ${}^{L(i),D} \in \mathbb{Q}_p[[X]] \otimes D_{cris}$ such that

$$(\langle \chi \rangle^{s} \tau^{i})(L^{(i),D}(\gamma-1)) = L_{p}(E, \langle \chi \rangle^{s} \tau^{i}(D/.)).$$

- When *E* has good ordinary or bad multiplicative reduction at *p*. By Weierstrass's preparation theorem the series $L^{(i),D}$ can be written $p^{\mu}(X^{\lambda} + pG(X))$ up to a *p*-adic unit, where $G(X) \in \mathbb{Z}_p[X]$. The function returns $[\lambda, \mu]$.
- When E has good supersingular reduction, we define a sequence of polynomials P_n in $\mathbb{Q}_p[X]$ of degree $< p^n$ (and bounded denominators), such that

 ${}^{L(i),D} = P_n \varphi^{n+1} \omega_E - \xi_n P_{n-1} \varphi^{n+2} \omega_E mod((1+X)^{p^n} - 1) \mathbb{Q}_p[X] \otimes D_{cris},$ where : math : ' $\xi_n = polcyclo(p^n, 1+X)$ '.Let : math : ' λ_n, μ_n 'betheinvariants of : math : ' P_n '.We find that

- μ_n is nonnegative and decreasing for *n* of given parity hence μ_{2n} tends to a limit μ^+ and μ_{2n+1} tends to a limit μ^- (both conjecturally 0).
- there exists integers λ^+ , λ^- in \mathbb{Z} (denoted with a in the reference below) such that

$$\lim_{n \to oo} \lambda_{2n} + 1/(p+1) = \lambda^+ and \lim_{n \to oo} \lambda_{2n+1} + p/(p+1) = \lambda^-.$$

The function returns : math : '[[\lambda^+, \lambda^-], [\mu^+, \mu^-]]'.

Reference: B. Perrin-Riou, Arithmétique des courbes elliptiques à réduction supersinguli\`ere en *p*, *Experimental Mathematics*, **12**, 2003, pp. 155-186.

ellpadiclog(p, n, P)

Given E defined over $K = \mathbb{Q}$ or \mathbb{Q}_p and P = [x, y] on E(K) in the kernel of reduction mod p, let t(P) = -x/y be the formal group parameter; this function returns L(t), where L denotes the formal logarithm (mapping the formal group of E to the additive formal group) attached to the canonical invariant differential: $dL = dx/(2y+a_1x+a_3)$.

```
? E = ellinit([0,0,1,-4,2]); P = [-2,1];
? ellpadiclog(E,2,10,P)
%2 = 2 + 2^3 + 2^8 + 2^9 + 2^10 + 0(2^11)
? E = ellinit([17,42]);
? p=3; Ep = ellinit(E,p); \\ E mod p
? P=[114,1218]; ellorder(Ep,P) \\ the order of P on (E mod p) is 2
%5 = 2
? Q = ellmul(E,P,2) \\ we need a point of the form 2*P
%6 = [200257/7056, 90637343/592704]
? ellpadiclog(E,3,10,Q)
%7 = 3 + 2*3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 2*3^8 + 3^9 + 2*3^10 + 0(3^11)
```

ellpadicregulator(p, n, S)

Let E/\mathbb{Q} be an elliptic curve. Return the determinant of the Gram matrix of the vector of points $S = (S_1, ..., S_r)$ with respect to the "canonical" cyclotomic *p*-adic height on *E*, given to *n* (*p*-adic) digits.

When E has ordinary reduction at p, this is the expected Gram deteterminant in \mathbb{Q}_p .

In the case of supersingular reduction of E at p, the definition requires care: the regulator R is an element of $D := H_{dR}^1(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$, which is a two-dimensional \mathbb{Q}_p -vector space spanned by ω and $\eta = x\omega$ (which are defined over \mathbb{Q}) or equivalently but now over \mathbb{Q}_p by ω and $F\omega$ where F is the Frobenius endomorphism on D as defined in ellpadicfrobenius. On D we define the cyclotomic height $h_E = f\omega + g\eta$ (see ellpadicheight) and a canonical alternating bilinear form $[.,.]_D$ such that $[\omega,\eta]_D = 1$.

For any $\nu \in D$, we can define a height $h_{\nu} := [h_E, \nu]_D$ from $E(\mathbb{Q})$ to \mathbb{Q}_p and $\langle .., \cdot \rangle_{\nu}$ the attached bilinear form. In particular, if $h_E = f\omega + g\eta$, then $h_{\eta} = [h_E, \eta]_D = f$ and $h_{\omega} = [h_E, \omega]_D = -g$ hence $h_E = h_{\eta}\omega - h_{\omega}\eta$. Then, R is the unique element of D such that

$$[\omega, \nu]_D^{r-1}[R, \nu]_D = \det(\langle S_i, S_j \rangle_{\nu})$$

for all $\nu \in D$ not in $\mathbb{Q}_p \omega$. The ellpadic regulator function returns R in the basis $(\omega, F\omega)$, which was chosen so that p-adic BSD conjectures are easy to state, see ellpadicbsd.

Note that by definition

$$[R,\eta]_D = \det(\langle S_i, S_j \rangle_\eta)$$

and

$$[R, \omega + \eta]_D = \det(\langle S_i, S_j \rangle_{\omega + \eta}).$$

ellpadics2(p, n)

If p > 2 is a prime and E/\mathbb{Q} is an elliptic curve with ordinary good reduction at p, returns the slope of the unit eigenvector of ellpadicfrobenius(E,p,n), i.e., the action of Frobenius φ on the crystalline module $D_p(E) = \mathbb{Q}_p \otimes H^1_{dR}(E/\mathbb{Q})$ in the basis of the given model $(\omega, \eta = x\omega)$, where ω is the invariant differential $dx/(2y + a_1x + a_3)$. In other words, $\eta + s_2\omega$ is an eigenvector for the unit eigenvalue of φ .

? e=ellinit([17,42]); ? ellpadics2(e,13,4) %2 = 10 + 2*13 + 6*13*3 + 0(13*4)

This slope is the unique $c \in 3^{-1}\mathbb{Z}_p$ such that the odd solution $\sigma(t) = t + O(t^2)$ of

$$-d((1)/(\sigma)(d\sigma)/(\omega)) = (x(t) + c)\omega$$

is in $t\mathbb{Z}_p[[t]]$.

It is equal to $b_2/12 - E_2/12$ where E_2 is the value of the Katz *p*-adic Eisenstein series of weight 2 on (E, ω) . This is used to construct a canonical *p*-adic height when *E* has good ordinary reduction at *p* as follows

s2 = ellpadics2(E,p,n); h(E,p,n, P, s2) = ellpadicheight(E, [p,[1,-s2]],n, P);

Since s_2 does not depend on the point *P*, we compute it only once.

ellperiods(flag, precision)

Let w describe a complex period lattice ($w = [w_1, w_2]$ or an ellinit structure). Returns normalized periods $[W_1, W_2]$ generating the same lattice such that $\tau := W_1/W_2$ has positive imaginary part and lies in the standard fundamental domain for $SL_2(\mathbb{Z})$.

If flag = 1, the function returns $[[W_1, W_2], [\eta_1, \eta_2]]$, where η_1 and η_2 are the quasi-periods attached to $[W_1, W_2]$, satisfying $\eta_2 W_1 - \eta_1 W_2 = 2i\pi$.

The output of this function is meant to be used as the first argument given to ellwp, ellzeta, ellsigma or elleisnum. Quasi-periods are needed by ellzeta and ellsigma only.

```
? L = ellperiods([1,I],1);
? [w1,w2] = L[1]; [e1,e2] = L[2];
? e2*w1 - e1*w2
%3 = 6.2831853071795864769252867665590057684*I
```

```
? ellzeta(L, 1/2 + 2*I)
%4 = 1.5707963... - 6.283185307...*I
? ellzeta([1,I], 1/2 + 2*I) \\ same but less efficient
%4 = 1.5707963... - 6.283185307...*I
```

ellpointtoz(P, precision)

If $E/\mathbb{C} \ \mathbb{C}/\Lambda$ is a complex elliptic curve ($\Lambda = E.omega$), computes a complex number z, well-defined modulo the lattice Λ , corresponding to the point P; i.e. such that $P = [\wp_{\Lambda}(z), \wp'_{\Lambda}(z)]$ satisfies the equation

$$y^2 = 4x^3 - g_2x - g_3,$$

where g_2 , g_3 are the elliptic invariants.

If E is defined over \mathbb{R} and $P \in E(\mathbb{R})$, we have more precisely, $0 \leq \Re(t) < w1$ and $0 \le \Im(w2)$, where (w1, w2) are the real and complex periods of E.

If E is defined over a general number field, the function returns the values corresponding to the various complex embeddings of the curve and of the point, in the same order as E.nf.roots:

```
? E=ellinit([-22032-15552*x,0], nfinit(x^2-2));
? P=[-72*x-108,0];
? ellisoncurve(E,P)
%3 = 1
? ellpointtoz(E,P)
%4 = [-0.52751724240790530394437835702346995884*I,
-0.090507650025885335533571758708283389896*I]
? E.nf.roots
%5 = [-1.4142135623730950488016887242096980786, \\ x-> -sqrt(2)
1.4142135623730950488016887242096980786] \\ x-> sqrt(2)
```

If E/\mathbb{Q}_p has multiplicative reduction, then E/\mathbb{Q}_p is analytically isomorphic to $\mathbb{Q}_p^*/q^{\mathbb{Z}}$ (Tate curve) for some *p*-adic integer *q*. The behavior is then as follows:

- If the reduction is split $(E.tate[2] \text{ is a t_PADIC})$, we have an isomorphism $\phi : E(\mathbb{Q}_p) \mathbb{Q}_p^*/q^{\mathbb{Z}}$ and the function returns $\phi(P) \in \mathbb{Q}_p$.
- If the reduction is *not* split (E.tate[2]) is a t_POLMOD, we only have an isomorphism $\phi : E(K) K^*/q^{\mathbb{Z}}$ over the unramified quadratic extension K/\mathbb{Q}_p . In this case, the output $\phi(P) \in K$ is a t_POLMOD.

```
? E = ellinit([0,-1,1,0,0], 0(11^5)); P = [0,0];
? [u2,u,q] = E.tate; type(u) \\ split multiplicative reduction
%2 = "t_PADIC"
? ellmul(E, P, 5) \\ P has order 5
%3 = [0]
```

```
? z = ellpointtoz(E, [0,0])
%4 = 3 + 11^2 + 2*11^3 + 3*11^4 + 6*11^5 + 10*11^6 + 8*11^7 + 0(11^8)
? z^5
%5 = 1 + 0(11^9)
? E = ellinit(ellfromj(1/4), 0(2^6)); x=1/2; y=ellordinate(E,x)[1];
? z = ellpointtoz(E, [x,y]); \\ t_POLMOD of t_POL with t_PADIC coeffs
? liftint(z) \\ lift all p-adics
%8 = Mod(8*u + 7, u^2 + 437)
```

ellpow(*z*, *n*)

Deprecated alias for ellmul.

ellratpoints(h, flag)

E being an integral model of elliptic curve, return a vector containing the affine rational points on the curve of naive height less than *h*. If flag = 1, stop as soon as a point is found; return either an empty vector or a vector containing a single point. See hyperellratpoints for how *h* can be specified.

```
? E=ellinit([-25,1]);
? ellratpoints(E,10)
%2 = [[-5,1],[-5,-1],[-3,7],[-3,-7],[-1,5],[-1,-5],
[0,1],[0,-1],[5,1],[5,-1],[7,13],[7,-13]]
? ellratpoints(E,10,1)
%3 = [[-5,1]]
```

ellrootno(p)

E being an ell structure over \mathbb{Q} as output by ellinit, this function computes the local root number of its *L*-series at the place *p* (at the infinite place if *p* = 0). If *p* is omitted, return the global root number and in this case the curve can also be defined over a number field.

Note that the global root number is the sign of the functional equation and conjecturally is the parity of the rank of the Mordell-Weil group. The equation for E needs not be minimal at p, but if the model is already minimal the function will run faster.

ellsea(tors)

Let E be an *ell* structure as output by ellinit, defined over a finite field \mathbb{F}_q . This low-level function computes the order of the group $E(\mathbb{F}_q)$ using the SEA algorithm; compared to the high-level function ellcard, which includes SEA among its choice of algorithms, the tors argument allows to speed up a search for curves having almost prime order and whose quadratic twist may also have almost prime order. When tors is set to a nonzero value, the function returns 0 as soon as it detects that the order has a small prime factor not dividing tors; SEA considers modular polynomials of increasing prime degree ℓ and we return 0 as soon as we hit an ℓ (coprime to tors) dividing $\#E(\mathbb{F}_q)$:

```
? ellsea(ellinit([1,1], 2^56+3477), 1)
%1 = 72057594135613381
? forprime(p=2^128,oo, q = ellcard(ellinit([1,1],p)); if(isprime(q),break))
time = 6,571 ms.
? forprime(p=2^128,oo, q = ellsea(ellinit([1,1],p),1);if(isprime(q),break))
time = 522 ms.
```

In particular, set tors to 1 if you want a curve with prime order, to 2 if you want to allow a cofactor which is a power of two (e.g. for Edwards's curves), etc. The early exit on bad curves yields a massive speedup compared to running the cardinal algorithm to completion.

When tors is negative, similar checks are performed for the quadratic twist of the curve.

The following function returns a curve of prime order over \mathbb{F}_p .

```
cryptocurve(p) =
{
  while(1,
  my(E, N, j = Mod(random(p), p));
  E = ellinit(ellfromj(j));
  N = ellsea(E, 1); if (!N, continue);
  if (isprime(N), return(E));
  \\ try the quadratic twist for free
  if (isprime(2*p+2 - N), return(ellinit(elltwist(E))));
  );
}
? p = randomprime([2^255, 2^256]);
? E = cryptocurve(p); \\ insist on prime order
%2 = 47,447ms
```

The same example without early abort (using ellcard(E) instead of ellsea(E, 1)) runs for about 5 minutes before finding a suitable curve.

The availability of the seadata package will speed up the computation, and is strongly recommended. The generic function ellcard should be preferred when you only want to compute the cardinal of a given curve without caring about it having almost prime order:

- If the characteristic is too small ($p \le 7$) or the field cardinality is tiny ($q \le 523$) the generic algorithm ellcard is used instead and the tors argument is ignored. (The reason for this is that SEA is not implemented for $p \le 7$ and that if $q \le 523$ it is likely to run into an infinite loop.)
- If the field cardinality is smaller than about 2^{50} , the generic algorithm will be faster.
- Contrary to ellcard, ellsea does not store the computed cardinality in E.

ellsearch()

This function finds all curves in the elldata database satisfying the constraint defined by the argument N:

- if N is a character string, it selects a given curve, e.g. "11a1", or curves in the given isogeny class, e.g. "11a", or curves with given conductor, e.g. "11";
- if N is a vector of integers, it encodes the same constraints as the character string above, according to the ellconvertname correspondance, e.g. [11,0,1] for "11a1", [11,0] for "11a" and [11] for "11";
- if N is an integer, curves with conductor N are selected.

If N codes a full curve name, for instance "11a1" or [11,0,1], the output format is $[N, [a_1, a_2, a_3, a_4, a_6], G]$ where $[a_1, a_2, a_3, a_4, a_6]$ are the coefficients of the Weierstrass equation of the curve and G is a \mathbb{Z} -basis of the free part of the Mordell-Weil group attached to the curve.

? ellsearch("11a3")
%1 = ["11a3", [0, -1, 1, 0, 0], []]
? ellsearch([11,0,3])
%2 = ["11a3", [0, -1, 1, 0, 0], []]

If N is not a full curve name, then the output is a vector of all matching curves in the above format:

```
? ellsearch("11a")
%1 = [["11a1", [0, -1, 1, -10, -20], []],
["11a2", [0, -1, 1, -7820, -263580], []],
["11a3", [0, -1, 1, 0, 0], []]]
```

? ellsearch("11b")	
%2 = []	

ellsigma(z, flag, precision)

Computes the value at z of the Weierstrass σ function attached to the lattice L as given by ellperiods(, 1): including quasi-periods is useful, otherwise there are recomputed from scratch for each new z.

$$\sigma(z,L) = z \prod_{\omega \in L^*} (1 - (z)/(\omega)) e^{(z)/(\omega) + (z^2)/(2\omega^2)}.$$

It is also possible to directly input $L = [\omega_1, \omega_2]$, or an elliptic curve E as given by elliptic (L = E.omega).

```
? w = ellperiods([1,I], 1);
? ellsigma(w, 1/2)
%2 = 0.47494937998792065033250463632798296855
? E = ellinit([1,0]);
? ellsigma(E) \\ at 'x, implicitly at default seriesprecision
%4 = x + 1/60*x^5 - 1/10080*x^9 - 23/259459200*x^13 + O(x^17)
```

If flag = 1, computes an arbitrary determination of $\log(\sigma(z))$.

ellsub(*z1*, *z2*)

Difference of the points z1 and z2 on the elliptic curve corresponding to E.

elltamagawa()

The object E being an elliptic curve over a number field, returns the global Tamagawa number of the curve (including the factor at infinite places).

? e = ellinit([1, -1, 1, -3002, 63929]); \\ curve "90c6" from elldata ? elltamagawa(e) %2 = 288 ? [elllocalred(e,p)[4] | p<-[2,3,5]] %3 = [6, 4, 6] ? vecprod(%) \\ since e.disc > 0 the factor at infinity is 2 %4 = 144

elltaniyama(serprec)

Computes the modular parametrization of the elliptic curve E/\mathbb{Q} , where E is an ell structure as output by ellinit. This returns a two-component vector [u, v] of power series, given to n significant terms (seriesprecision by default), characterized by the following two properties. First the point (u, v) satisfies the equation of the elliptic curve. Second, let N be the conductor of E and $\Phi : X_0(N) \to E$ be a modular parametrization; the pullback by Φ of the Néron differential $du/(2v + a_1u + a_3)$ is equal to $2i\pi f(z)dz$, a holomorphic differential form. The variable used in the power series for u and v is x, which is implicitly understood to be equal to $\exp(2i\pi z)$.

The algorithm assumes that E is a *strong* Weil curve and that the Manin constant is equal to 1: in fact, $f(x) = \sum_{n>0} ellak(E, n)x^n$.

elltatepairing(P, Q, m)

Let *E* be an elliptic curve defined over a finite field *k* and $m \ge 1$ be an integer. This function computes the (nonreduced) Tate pairing of the points *P* and *Q* on *E*, where *P* is an *m*-torsion point. More precisely, let $f_{m,P}$ denote a Miller function with divisor $m[P] - m[O_E]$; the algorithm returns $f_{m,P}(Q) \in k^*/(k^*)^m$.

elltors()

If E is an elliptic curve defined over a number field or a finite field, outputs the torsion subgroup of E as a 3-component vector [t,v1,v2], where t is the order of the torsion group, v1 gives the structure of the torsion group as a product of cyclic groups (sorted by decreasing order), and v2 gives generators for these cyclic groups. E must be an ell structure as output by ellinit.

? E = ellinit([-1,0]); ? elltors(E) %1 = [4, [2, 2], [[0, 0], [1, 0]]]

Here, the torsion subgroup is isomorphic to $\mathbb{Z}/2\mathbb{Z}x\mathbb{Z}/2\mathbb{Z}$, with generators [0,0] and [1,0].

elltwist(P)

Returns the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of the twist of the elliptic curve E by the quadratic extension of the coefficient ring defined by P (when P is a polynomial) or quadpoly(P) when P is an integer. If E is defined over a finite field, then P can be omitted, in which case a random model of the unique nontrivial twist is returned. If E is defined over a number field, the model should be replaced by a minimal model (if one exists).

Example: Twist by discriminant -3:

? elltwist(ellinit([0,a2,0,a4,a6]),-3) %1 = [0,-3*a2,0,9*a4,-27*a6]

Twist by the Artin-Schreier extension given by $x^2 + x + T$ in characteristic 2:

```
? lift(elltwist(ellinit([a1,a2,a3,a4,a6]*Mod(1,2)),x^2+x+T))
%1 = [a1,a2+a1^2*T,a3,a4,a6+a3^2*T]
```

Twist of an elliptic curve defined over a finite field:

```
? E=ellinit([1,7]*Mod(1,19));lift(elltwist(E))
%1 = [0,0,0,11,12]
```

ellweilcurve(ms)

If E' is an elliptic curve over \mathbb{Q} , let $L_{E'}$ be the sub- \mathbb{Z} -module of $\operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, \mathbb{Q})$ attached to E' (It is given by x[3] if [M, x] = msfromell(E').)

On the other hand, if N is the conductor of E and f is the modular form for $\Gamma_0(N)$ attached to E, let L_f be the lattice of the f-component of $\operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, \mathbb{Q})$ given by the elements ϕ such that $\phi(0, \gamma^{-1}0) \in \mathbb{Z}$ for all $\gamma \in \Gamma_0(N)$ (see mslattice).

Let E' run through the isomorphism classes of elliptic curves isogenous to E as given by ellisomat (and in the same order). This function returns a pair [vE, vS] where vE contains minimal models for the E' and vS contains the list of Smith invariants for the lattices $L_{E'}$ in L_f . The function also accepts the output of ellisomat, i.e. the isogeny class. If the optional argument ms is present, it contains the output of msfromell(vE, 0), i.e. the new modular symbol space M of level N and a vector of triples $[x^+, x^-, L]$ attached to each curve E'.

In particular, the strong Weil curve amongst the curves isogenous to E is the one whose Smith invariants are [c, c], where c is the Manin constant, conjecturally equal to 1.

```
? E = ellinit("11a3");
? [vE, vS] = ellweilcurve(E);
? [n] = [ i | i<-[1..#vS], vS[i]==[1,1] ] \\ lattice with invariant [1,1]
%3 = [2]
? ellidentify(vE[n]) \\ ... corresponds to strong Weil curve
%4 = [["11a1", [0, -1, 1, -10, -20], []], [1, 0, 0, 0]]
```

? [vE, vS] = ellweilcurve(E, &ms); \\ vE,vS are as above ? [M, vx] = ms; msdim(M) \\ ... but ms contains more information %6 = 3 ? #vx %7 = 3 ? vx[1] %8 = [[1/25, -1/10, -1/10]~, [0, 1/2, -1/2]~, [1/25,0; -3/5,1; 2/5,-1]] ? forell(E, 11,11, print(msfromell(ellinit(E[1]), 1)[2])) [1/5, -1/2, -1/2]~ [1, -5/2, -5/2]~ [1/25, -1/10, -1/10]~

The last example prints the modular symbols x^+ in M^+ attached to the curves 11a1, 11a2 and 11a3.

ellweilpairing(P, Q, m)

Let *E* be an elliptic curve defined over a finite field and $m \ge 1$ be an integer. This function computes the Weil pairing of the two *m*-torsion points *P* and *Q* on *E*, which is an alternating bilinear map. More precisely, let $f_{m,R}$ denote a Miller function with divisor $m[R] - m[O_E]$; the algorithm returns the *m*-th root of unity

$$\varepsilon(P,Q)^m \cdot f_{m,P}(Q) / f_{m,Q}(P),$$

where f(R) is the extended evaluation of f at the divisor $[R] - [O_E]$ and $\varepsilon(P, Q) \in 1$ is given by Weil reciprocity: $\varepsilon(P, Q) = 1$ if and only if P, Q, O_E are not pairwise distinct.

ellwp(z, flag, precision)

Computes the value at z of the Weierstrass \wp function attached to the lattice w as given by ellperiods. It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by elliptic (w = E.omega).

```
? w = ellperiods([1,I]);
? ellwp(w, 1/2)
%2 = 6.8751858180203728274900957798105571978
? E = ellinit([1,1]);
? ellwp(E, 1/2)
%4 = 3.9413112427016474646048282462709151389
```

One can also compute the series expansion around z = 0:

```
? E = ellinit([1,0]);
? ellwp(E) \\ 'x implicitly at default seriesprecision
%5 = x^-2 - 1/5*x^2 + 1/75*x^6 - 2/4875*x^10 + 0(x^14)
? ellwp(E, x + 0(x^12)) \\ explicit precision
%6 = x^-2 - 1/5*x^2 + 1/75*x^6 + 0(x^9)
```

Optional *flag* means 0 (default): compute only $\wp(z)$, 1: compute $[\wp(z), \wp'(z)]$.

For instance, the Dickson elliptic functions sm and sn can be implemented as follows

```
smcm(z) =
{ my(a, b, E = ellinit([0,-1/(4*27)])); \\ ell. invariants (g2,g3)=(0,1/27)
[a,b] = ellwp(E, z, 1);
[6*a / (1-3*b), (3*b+1)/(3*b-1)];
}
? [s,c] = smcm(0.5);
```

```
? s
%2 = 0.4898258757782682170733218609
? c
%3 = 0.9591820206453842491187464098
? s^3+c^3
%4 = 1.0000000000000000000000
? smcm('x + 0('x^11))
%5 = [x - 1/6*x^4 + 2/63*x^7 - 13/2268*x^10 + 0(x^11),
1 - 1/3*x^3 + 1/18*x^6 - 23/2268*x^9 + 0(x^10)]
```

ellxn(n, v)

For any affine point P = (t, u) on the curve E, we have

$$[n]P = (\phi_n(P)\psi_n(P):\omega_n(P):\psi_n(P)^3)$$

for some ϕ_n, ω_n, ψ_n in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][t, u]$ modulo the curve equation. This function returns a pair [A, B] of polynomials in $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][v]$ such that $[A(t), B(t)] = [\phi_n(P), \psi_n(P)^2]$ in the function field of E, whose quotient give the abscissa of [n]P. If P is an n-torsion point, then B(t) = 0.

```
? E = ellinit([17,42]); [t,u] = [114,1218];
? T = ellxn(E, 2, 'X)
%2 = [X^4 - 34*X^2 - 336*X + 289, 4*X^3 + 68*X + 168]
? [a,b] = subst(T,'X,t);
%3 = [168416137, 5934096]
? a / b == ellmul(E, [t,u], 2)[1]
%4 = 1
```

ellzeta(z, precision)

Computes the value at z of the Weierstrass ζ function attached to the lattice w as given by ellperiods(, 1): including quasi-periods is useful, otherwise there are recomputed from scratch for each new z.

$$\zeta(z,L) = (1)/(z) + z^2 \sum_{\omega \in L^*} (1)/(\omega^2(z-\omega)).$$

It is also possible to directly input $w = [\omega_1, \omega_2]$, or an elliptic curve E as given by elliptic (w = E.omega). The quasi-periods of ζ , such that

$$\zeta(z + a\omega_1 + b\omega_2) = \zeta(z) + a\eta_1 + b\eta_2$$

for integers a and b are obtained as $\eta_i = 2\zeta(\omega_i/2)$. Or using directly elleta.

```
? w = ellperiods([1,I],1);
? ellzeta(w, 1/2)
%2 = 1.5707963267948966192313216916397514421
? E = ellinit([1,0]);
? ellzeta(E, E.omega[1]/2)
%4 = 0.84721308479397908660649912348219163647
```

One can also compute the series expansion around z = 0 (the quasi-periods are useless in this case):

```
? E = ellinit([0,1]);

? ellzeta(E) \setminus at 'x, implicitly at default seriesprecision

%4 = x^{-1} + 1/35*x^{5} - 1/7007*x^{11} + 0(x^{15})

? ellzeta(E, x + 0(x^{20})) \setminus explicit precision

%5 = x^{-1} + 1/35*x^{5} - 1/7007*x^{11} + 1/1440257*x^{17} + 0(x^{18})
```

ellztopoint(z, precision)

E being an *ell* as output by ellinit, computes the coordinates [x, y] on the curve *E* corresponding to the complex or *p*-adic parameter *z*. Hence this is the inverse function of ellpointtoz.

• If E is defined over a p-adic field and has multiplicative reduction, then z is understood as an element on the Tate curve $\bar{Q}_{p}^{*}/q^{\mathbb{Z}}$.

```
? E = ellinit([0,-1,1,0,0], 0(11^5));
? [u2,u,q] = E.tate; type(u)
%2 = "t_PADIC" \\ split multiplicative reduction
? z = ellpointtoz(E, [0,0])
%3 = 3 + 11^2 + 2*11^3 + 3*11^4 + 6*11^5 + 10*11^6 + 8*11^7 + 0(11^8)
? ellztopoint(E,z)
%4 = [0(11^9), 0(11^9)]
? E = ellinit(ellfromj(1/4), 0(2^6)); x=1/2; y=ellordinate(E,x)[1];
? z = ellpointtoz(E,[x,y]); \\ nonsplit: t_POLMOD with t_PADIC coefficients
? P = ellztopoint(E, z);
? P[1] \\ y coordinate is analogous, more complicated
%8 = Mod(0(2^4)*x + (2^-1 + 0(2^5)), x^2 + (1 + 2^2 + 2^4 + 2^5 + 0(2^7)))
```

• If E is defined over the complex numbers (for instance over \mathbb{Q}), z is understood as a complex number in \mathbb{C}/Λ_E . If the short Weierstrass equation is $y^2 = 4x^3 - g_2x - g_3$, then [x, y] represents the Weierstrass \wp -function and its derivative. For a general Weierstrass equation we have

 $x = \wp(z) - b_2/12, y = \wp'(z)/2 - (a_1x + a_3)/2.$ If : math : 'z'isinthelatticedefining : math : 'E'over : math : 'C', there sultist he point at in finity : math : '[0]'.

erfc(precision)

Complementary error function, analytic continuation of $(2/\sqrt{\pi}) \int_x^o oe^{-t^2} dt = sign(x)incgam(1/2, x^2)/\sqrt{\pi}$ for real x! = 0. The latter expression extends the function definition from real x to complex x with positive real part (or zero real part and positive imaginary part). This is extended to the whole complex plane by the functional equation erfc(-x) = 2 - erfc(x).

errname()

Returns the type of the error message E as a string.

```
? iferr(1 / 0, E, print(errname(E)))
e_INV
? ?? e_INV
[...]
* "e_INV". Tried to invert a noninvertible object x in function s.
[...]
```

eta(flag, precision)

Variants of Dedekind's η function. If flag = 0, return $\prod_{n=1}^{o} o(1 - q^n)$, where q depends on x in the following way:

- $q = e^{2i\pi x}$ if x is a *complex number* (which must then have positive imaginary part); notice that the factor $q^{1/24}$ is missing!
- q = x if x is a t_PADIC, or can be converted to a *power series* (which must then have positive valuation).

If *flag* is nonzero, x is converted to a complex number and we return the true η function, $q^{1/24} \prod_{n=1}^{o} o(1-q^n)$, where $q = e^{2i\pi x}$.

eulerphi()

Euler's ϕ (totient) function of the integer ||x||, in other words $||(\mathbb{Z}/x\mathbb{Z})^*||$.

```
? eulerphi(40)
%1 = 16
```

According to this definition we let $\phi(0) := 2$, since $\mathbb{Z}^* = -1, 1$; this is consistent with znstar(0): we have znstar:math:`(n).no = eulerphi(n)` for all $n \in \mathbb{Z}$.

exp(precision)

Exponential of x. p-adic arguments with positive valuation are accepted.

expm1(precision)

Return $\exp(x) - 1$, computed in a way that is also accurate when the real part of x is near 0. A naive direct computation would suffer from catastrophic cancellation; PARI's direct computation of $\exp(x)$ alleviates this well known problem at the expense of computing $\exp(x)$ to a higher accuracy when x is small. Using expm1 is recommended instead:

```
? default(realprecision, 10000); x = 1e-100;
? a = expm1(x);
time = 4 ms.
? b = exp(x)-1;
time = 4 ms.
? default(realprecision, 10040); x = 1e-100;
? c = expm1(x); \\ reference point
? abs(a-c)/c \\ relative error in expm1(x)
%7 = 1.4027986153764843997 E-10019
? abs(b-c)/c \\ relative error in exp(x)-1
%8 = 1.7907031188259675794 E-9919
```

As the example above shows, when x is near 0, expm1 is more accurate than exp(x)-1.

exponent()

When x is a t_REAL, the result is the binary exponent e of x. For a nonzero x, this is the unique integer e such

that $2^e \le ||x|| < 2^{e+1}$. For a real 0, this returns the PARI exponent e attached to x (which may represent any floating-point number less than 2^e in absolute value).

```
? exponent(Pi)
%1 = 1
? exponent(4.0)
%2 = 2
? exponent(0.0)
%3 = -128
? default(realbitprecision)
%4 = 128
```

This definition extends naturally to nonzero integers, and the exponent of an exact 0 is -oo by convention.

For convenience, we *define* the exponent of a t_FRAC a/b as the difference of exponent(a) and exponent(b); note that, if e' denotes the exponent of :math: `a/b * 1.0`, then the exponent e we return is either e' or e' + 1, thus 2^{e+1} is an upper bound for ||a/b||.

```
? [ exponent(9), exponent(10), exponent(9/10), exponent(9/10*1.) ]
%5 = [3, 3, 0, -1]
```

For a PARI object of type t_COMPLEX, t_POL, t_SER, t_VEC, t_COL, t_MAT this returns the largest exponent found among the components of x. Hence 2^{e+1} is a quick upper bound for the sup norm of real matrices or polynomials; and $2^{e+(3/2)}$ for complex ones.

```
? exponent(3*x^2 + 15*x - 100)
%5 = 6
? exponent(0)
%6 = -oo
```

factor(D)

Factor x over domain D; if D is omitted, it is determined from x. For instance, if x is an integer, it is factored in \mathbb{Z} , if it is a polynomial with rational coefficients, it is factored in $\mathbb{Q}[x]$, etc., see below for details. The result is a two-column matrix: the first contains the irreducibles dividing x (rational or Gaussian primes, irreducible polynomials), and the second the exponents. By convention, 0 is factored as 0^1 .

:math: x in mathbb{Q}. See factorint for the algorithms used. The factorization includes the unit -1 when x < 0 and all other factors are positive; a denominator is factored with negative exponents. The factors are sorted in increasing order.

```
? factor(-7/106)
%1 =
[-1 1]
[ 2 -1]
[ 7 1]
[53 -1]
```

By convention, 1 is factored as matrix(0,2) (the empty factorization, printed as [;]).

Large rational "primes" > 2^{64} in the factorization are in fact *pseudoprimes* (see ispseudoprime), a priori not rigorously proven primes. Use isprime to prove primality of these factors, as in

? fa = factor(2^2^7 + 1)
%2 =
[59649589127497217 1]
[5704689200685129054721 1]
? isprime(fa[,1])
%3 = [1, 1]~ \\ both entries are proven primes

Another possibility is to globally set the default factor_proven, which will perform a rigorous primality proof for each pseudoprime factor but will slow down PARI.

A t_INT argument D can be added, meaning that we only trial divide by all primes p < D and the addprimes entries, then skip all expensive factorization methods. The limit D must be nonnegative. In this case, one entry in the factorization may be a composite number: all factors less than D^2 and primes from the addprimes table are actual primes. But (at most) one entry may not verify this criterion, and it may be prime or composite: it is only known to be coprime to all other entries and not a pure power..

? factor(2^2^7 +1, 10^5)
%4 =
[340282366920938463463374607431768211457 1]

Deprecated feature. Setting D = 0 is the same as setting it to *primelimit* + 1.

This routine uses trial division and perfect power tests, and should not be used for huge values of D (at most 10^9 , say): factorint(, 1 + 8) will in general be faster. The latter does not guarantee that all small prime factors are found, but it also finds larger factors and in a more efficient way.

```
? F = (2^2^7 + 1) * 1009 * (10^5+3); factor(F, 10^5) \\ fast, incomplete
time = 0 \text{ ms}.
%5 =
[1009 1]
[34029257539194609161727850866999116450334371 1]
? factor(F, 10^9) \\ slow
time = 3,260 ms.
%6 =
[1009 1]
[100003 1]
[340282366920938463463374607431768211457 1]
? factorint(F, 1+8) \\ much faster and all small primes were found
time = 8 \text{ ms}.
%7 =
[1009 1]
[100003 1]
[340282366920938463463374607431768211457 1]
? factor(F) \\ complete factorization
```

time = 60 ms. %8 = [1009 1] [100003 1] [59649589127497217 1] [5704689200685129054721 1]

Setting D = I will factor in the Gaussian integers $\mathbb{Z}[i]$:

:math:`x in mathbb{Q} (i).` The factorization is performed with Gaussian primes in $\mathbb{Z}[i]$ and includes Gaussian units in 1, *i*; factors are sorted by increasing norm. Except for a possible leading unit, the Gaussian factors are normalized: rational factors are positive and irrational factors have positive imaginary part (a canonical representa.

Unless factor_proven is set, large factors are actually pseudoprimes, not proven primes; a rational factor is prime if less than 2^{64} and an irrational one if its norm is less than 2^{64} .

? factor(5*I) %9 = [2 + I 1] [1 + 2*I 1]

One can force the factorization of a rational number by setting the domain D = I:

```
? factor(-5, I)
%10 =
[ I 1]
[ 2 + I 1]
[1 + 2*I 1]
? factorback(%)
%11 = -5
```

Univariate polynomials and rational functions. PARI can factor univariate polynomials in K[t]. The following base fields K are currently supported: \mathbb{Q} , \mathbb{R} , \mathbb{C} , \mathbb{Q}_p , finite fields and number fields. See factormod and factorff for the algorithms used over finite fields and nffactor for the algorithms over number fields. The irreducible factors are sorted by increasing degree and normalized: they are monic except when $K = \mathbb{Q}$ where they are primitive in $\mathbb{Z}[t]$.

The content is *not* included in the factorization, in particular factorback will in general recover the original x only up to multiplication by an element of K^* : when $K! = \mathbb{Q}$, this scalar is pollead(x) (since irreducible factors are monic); and when $K = \mathbb{Q}$ you can either ask for the \mathbb{Q} -content explicitly of use factorback:

? P = t^2 + 5*t/2 + 1; F = factor(P) %12 = [t + 2 1] [2*t + 1 1] ? content(P, 1) \\ Q-content

```
%13 = 1/2
? pollead(factorback(F)) / pollead(P)
%14 = 2
```

You can specify K using the optional "domain" argument D as follows

- $K = \mathbb{Q}$: D a rational number (t_INT or t_FRAC),
- $K = \mathbb{Z}/p\mathbb{Z}$ with p prime : D a t_INTMOD modulo p; factoring modulo a composite number is not supported.
- $K = \mathbb{F}_q$: D a t_FFELT encoding the finite field; you can also use a t_POLMOD of t_INTMOD modulo a prime p but this is usually less convenient;
- $K = \mathbb{Q}[X]/(T)$ a number field : D a t_POLMOD modulo T,
- $K = \mathbb{Q}(i)$ (alternate syntax for special case): D = I,
- $K = \mathbb{Q}(w)$ a quadratic number field (alternate syntax for special case): D a t_QUAD,
- $K = \mathbb{R} : D$ a real number (t_REAL); truncate the factorization at accuracy $\operatorname{precision}(D)$. If x is inexact and $\operatorname{precision}(x)$ is less than $\operatorname{precision}(D)$, then the precision of x is used instead.
- $K = \mathbb{C}$: D a complex number with a t_REAL component, e.g. I * 1.; truncate the factorization as for $K = \mathbb{R}$,
- K = Q_p : D a t_PADIC; truncate the factorization at p-adic accuracy padicprec(D), possibly less if x is inexact with insufficient p-adic accuracy;

```
? T = x^2+1;
? factor(T, 1); \\ over Q
? factor(T, Mod(1,3)) \\ over F_3
? factor(T, ffgen(ffinit(3,2,'t))^0) \\ over F_{3^2}
? factor(T, Mod(Mod(1,3), t^2+t+2)) \\ over F_{3^2}, again
? factor(T, 0(3^6)) \\ over Q_3, precision 6
? factor(T, 1.) \\ over R, current precision
? factor(T, I*1.) \\ over C
? factor(T, Mod(1, y^3-2)) \\ over Q(2^{1/3})
```

In most cases, it is possible and simpler to call a specialized variant rather than use the above scheme:

```
? factormod(T, 3) \\ over F_3
? factormod(T, [t<sup>2</sup>+t+2, 3]) \\ over F_{3^2}
? factormod(T, ffgen(3^2, 't)) \\ over F_{3^2}
? factorpadic(T, 3,6) \\ over Q_3, precision 6
? nffactor(y^3-2, T) \\ over Q(2^{1/3})
? polroots(T) \\ over C
? polrootsreal(T) \\ over R (real polynomial)
```

It is also possible to let the routine use the smallest field containing all coefficients, taking into account quotient structures induced by t_INTMOD s and t_POLMOD s (e.g. if a coefficient in $\mathbb{Z}/n\mathbb{Z}$ is known, all rational numbers encountered are first mapped to $\mathbb{Z}/n\mathbb{Z}$; different moduli will produce an error):

```
? T = x^2+1;
? factor(T); \\ over Q
? factor(T*Mod(1,3)) \\ over F_3
? factor(T*ffgen(ffinit(3,2,'t))^0) \\ over F_{3^2}
```

```
? factor(T*Mod(Mod(1,3), t^2+t+2)) \\ over F_{3^2}, again
? factor(T*(1 + 0(3^6)) \\ over Q_3, precision 6
? factor(T*1.) \\ over R, current precision
? factor(T*(1.+0.*I)) \\ over C
? factor(T*Mod(1, y^3-2)) \\ over Q(2^{1/3})
```

Multiplying by a suitable field element equal to $1 \in K$ in this way is error-prone and is not recommanded. Factoring existing polynomials with obvious fields of coefficients is fine, the domain argument D should be used instead ad hoc conversions.

Note on inexact polynomials. Polynomials with inexact coefficients (e.g. floating point or *p*-adic numbers) are first rounded to an exact representation, then factored to (potentially) infinite accuracy and we return a truncated approximation of that virtual factorization. To avoid pitfalls, we advise to only factor *exact* polynomials:

```
? factor(x^{2}-1+0(2^{2})) \\ rounded to x^{2} + 3, irreducible in Q_2
%1 =
[(1 + 0(2^{2}))*x^{2} + 0(2^{2})*x + (1 + 2 + 0(2^{2})) 1]
? factor(x^{2}-1+0(2^{3})) \\ rounded to x^{2} + 7, reducible !
%2 =
[ (1 + 0(2^{3}))*x + (1 + 2 + 0(2^{3})) 1]
[(1 + 0(2^{3}))*x + (1 + 2^{2} + 0(2^{3})) 1]
[(1 + 0(2^{3}))*x + (1 + 2^{2} + 0(2^{3})) 1]
? factor(x^{2}-1, 0(2^{2})) \\ no ambiguity now
%3 =
[ (1 + 0(2^{2}))*x + (1 + 0(2^{2})) 1]
[(1 + 0(2^{2}))*x + (1 + 2 + 0(2^{2})) 1]
```

Note about inseparable polynomials. Polynomials with inexact coefficients are considered to be squarefree: indeed, there exist a squarefree polynomial arbitrarily close to the input, and they cannot be distinguished at the input accuracy. This means that irreducible factors are repeated according to their apparent multiplicity. On the contrary, using a specialized function such as factorpadic with an *exact* rational input yields the correct multiplicity when the (now exact) input is not separable. Compare:

```
? factor(z<sup>2</sup> + 0(5<sup>2</sup>)))
%1 =
[(1 + 0(5<sup>2</sup>))*z + 0(5<sup>2</sup>) 1]
[(1 + 0(5<sup>2</sup>))*z + 0(5<sup>2</sup>) 1]
? factor(z<sup>2</sup>, 0(5<sup>2</sup>))
%2 =
[1 + 0(5<sup>2</sup>))*z + 0(5<sup>2</sup>) 2]
```

Multivariate polynomials and rational functions. PARI recursively factors *multivariate* polynomials in $K[t_1, ..., t_d]$ for the same fields K as above and the argument D is used in the same way to specify K. The irreducible factors are sorted by their main variable (least priority first) then by increasing degree.

```
? factor(x^2 + y^2, Mod(1,5))
%1 =
[ x + Mod(2, 5)*y 1]
```

[Mod(1, 5)*x + Mod(3, 5)*y 1] ? factor(x^2 + y^2, 0(5^2)) %2 = [(1 + 0(5^2))*x + (0(5^2)*y^2 + (2 + 5 + 0(5^2))*y + 0(5^2)) 1] [(1 + 0(5^2))*x + (0(5^2)*y^2 + (3 + 3*5 + 0(5^2))*y + 0(5^2)) 1] ? lift(%) %3 = [x + 7*y 1] [x + 18*y 1]

Note that the implementation does not really support inexact real fields (\mathbb{R} or \mathbb{C}) and usually misses factors even if the input is exact:

```
? factor(x^2 + y^2, I) \\ over Q(i)
%4 =
[x - I*y 1]
[x + I*y 1]
? factor(x^2 + y^2, I*1.) \\ over C
%5 =
[x^2 + y^2 1]
```

factorback(e)

Gives back the factored object corresponding to a factorization. The integer 1 corresponds to the empty factorization.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization, as produced with any factor command. A few examples:

```
? factor(12)
%1 =
[2 2]
[3 1]
? factorback(%)
%2 = 12
? factorback([2,3], [2,1]) \\ 2^3 * 3^1
%3 = 12
? factorback([5,2,3])
%4 = 30
```

factorcantor(p)

This function is obsolete, use factormod.
factorff(p, a)

Obsolete, kept for backward compatibility: use factormod.

factorint(flag)

Factors the integer n into a product of pseudoprimes (see ispseudoprime), using a combination of the Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers. The output is a two-column matrix as for factor: the first column contains the "prime" divisors of n, the second one contains the (positive) exponents.

By convention 0 is factored as 0^1 , and 1 as the empty factorization; also the divisors are by default not proven primes if they are larger than 2^{64} , they only failed the BPSW compositeness test (see ispseudoprime). Use isprime on the result if you want to guarantee primality or set the factor_proven default to 1. Entries of the private prime tables (see addprimes) are also included as is.

This gives direct access to the integer factoring engine called by most arithmetical functions. *flag* is optional; its binary digits mean 1: avoid MPQS, 2: skip first stage ECM (we may still fall back to it later), 4: avoid Rho and SQUFOF, 8: don't run final ECM (as a result, a huge composite may be declared to be prime). Note that a (strong) probabilistic primality test is used; thus composites might not be detected, although no example is known.

You are invited to play with the flag settings and watch the internals at work by using gp's debug default parameter (level 3 shows just the outline, 4 turns on time keeping, 5 and above show an increasing amount of internal details).

factormod(D, flag)

Factors the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T(y) an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[y]/(T)$ (as usual the main variable of T must have lower priority than the main variable of f);
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

The coefficients of f must be operation-compatible with the corresponding finite field. The result is a twocolumn matrix, the first column being the irreducible polynomials dividing f, and the second the exponents. By convention, the 0 polynomial factors as 0^1 ; a nonzero constant polynomial has empty factorization, a $0x^2$ matrix. The irreducible factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? factormod(x^2 + 1, 3) \\ over F_3
%1 =
[Mod(1, 3)*x^2 + Mod(1, 3) 1]
? liftall( factormod(x^2 + 1, [t^2+1, 3]) ) \\ over F_9
%2 =
[ x + t 1]
[x + 2*t 1]
\\ same, now letting GP choose a model
? T = ffinit(3,2,'t)
%3 = Mod(1, 3)*t^2 + Mod(1, 3)*t + Mod(2, 3)
? liftall( factormod(x^2 + 1, [T, 3]) )
%4 = \\ t is a root of T !
[ x + (t + 2) 1]
[x + (2*t + 1) 1]
```

```
? t = ffgen(t^2+Mod(1,3)); factormod(x^2 + t^0) \\ same using t_FFELT
%5 =
[ x + t 1]
[x + 2*t 1]
? factormod(x^2+Mod(1,3))
%6 =
[Mod(1, 3)*x^2 + Mod(1, 3) 1]
? liftall( factormod(x^2 + Mod(Mod(1,3), y^2+1)) )
%7 =
[ x + y 1]
[x + 2*y 1]
```

If flag is nonzero, outputs only the *degrees* of the irreducible polynomials (for example to compute an *L*-function). By convention, a constant polynomial (including the 0 polynomial) has empty factorization. The degrees appear in increasing order but need not correspond to the ordering with flag = 0 when multiplicities are present.

```
? f = x^3 + 2*x^2 + x + 2;
? factormod(f, 5) \\ (x+2)^2 * (x+3)
%1 =
[Mod(1, 5)*x + Mod(2, 5) 2]
[Mod(1, 5)*x + Mod(3, 5) 1]
? factormod(f, 5, 1) \\ (deg 1) * (deg 1)^2
%2 =
[1 1]
[1 2]
```

factormodDDF(D)

Distinct-degree factorization of the squarefree polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[x]/(T)$;
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

This is somewhat faster than full factorization. The coefficients of f must be operation-compatible with the corresponding finite field. The result is a two-column matrix:

- the first column contains monic (squarefree) pairwise coprime polynomials dividing f, all of whose irreducible factors have degree d;
- the second column contains the degrees of the irreducible factors.

The factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? f = (x^2 + 1) * (x^2-1);
? factormodSQF(f,3) \\ squarefree over F_3
```

```
%2 =
[Mod(1, 3)*x^4 + Mod(2, 3) 1]
? factormodDDF(f, 3)
%3 =
[Mod(1, 3)*x^2 + Mod(2, 3) 1] \setminus two degree 1 factors
[Mod(1, 3)*x^2 + Mod(1, 3) 2] \setminus irred of degree 2
? for(i=1,10^5,factormodDDF(f,3))
time = 424 \text{ ms}.
? for(i=1,10^5,factormod(f,3)) \\ full factorization is slower
time = 464 \text{ ms}.
? liftall( factormodDDF(x^2 + 1, [3, t^2+1]) ) \\ over F_9
%6 =
[x^2 + 1 \ 1] \setminus  product of two degree 1 factors
? t = ffgen(t^2+Mod(1,3)); factormodDDF(x^2 + t^0) \ same using t_FFELT
%7 =
[x^2 + 1 1]
? factormodDDF(x^2-Mod(1,3))
%8 =
[Mod(1, 3)*x^2 + Mod(2, 3) 1]
```

factormodSQF(D)

Squarefree factorization of the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[x]/(T)$;
- *D* a t_FFELT: factor over the attached field;
- D omitted: factor over the field of definition of f, which must be a finite field.

This is somewhat faster than full factorization. The coefficients of f must be operation-compatible with the corresponding finite field. The result is a two-column matrix:

- the first column contains monic squarefree pairwise coprime polynomials dividing f;
- the second column contains the power to which the polynomial in column 1 divides f;

The factors are ordered by increasing degree and the result is canonical: it will not change across multiple calls or sessions.

```
? f = (x^2 + 1)^3 * (x^2-1)^2;
? factormodSQF(f, 3) \\ over F_3
%1 =
[Mod(1, 3)*x^2 + Mod(2, 3) 2]
[Mod(1, 3)*x^2 + Mod(1, 3) 3]
? for(i=1,10^5,factormodSQF(f,3))
time = 192 ms.
```

```
(continued from previous page)
```

```
? for(i=1,10^5,factormod(f,3)) \\ full factorization is slower
time = 409 ms.
? liftall( factormodSQF((x^2 + 1)^3, [3, t^2+1]) ) \\ over F_9
%4 =
[x^2 + 1 3]
? t = ffgen(t^2+Mod(1,3)); factormodSQF((x^2 + t^0)^3) \\ same using t_FFELT
%5 =
[x^2 + 1 3]
? factormodSQF(x^8 + x^7 + x^6 + x^2 + x + Mod(1,2))
%6 =
[ Mod(1, 2)*x + Mod(1, 2) 2]
[Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2) 3]
```

factornf(*t*)

This function is obsolete, use nffactor.

factorization of the univariate polynomial x over the number field defined by the (univariate) polynomial t. x may have coefficients in \mathbb{Q} or in the number field. The algorithm reduces to factorization over \mathbb{Q} (Trager's trick). The direct approach of nffactor, which uses van Hoeij's method in a relative setting, is in general faster.

The main variable of t must be of *lower* priority than that of x (see priority (in the PARI manual)). However if nonrational number field elements occur (as polmods or polynomials) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t. For example

```
? factornf(x^2 + Mod(y, y^2+1), y^2+1);
? factornf(x^2 + y, y^2+1); \\ these two are OK
? factornf(x^2 + Mod(z,z^2+1), y^2+1)
*** at top-level: factornf(x^2+Mod(z,z
*** ^------
*** factornf: inconsistent data in rnf function.
? factornf(x^2 + z, y^2+1)
*** at top-level: factornf(x^2+z,y^2+1
*** ^------
*** factornf: incorrect variable in rnf function.
```

factorpadic(p, r)

p-adic factorization of the polynomial *pol* to precision *r*, the result being a two-column matrix as in factor. Note that this is not the same as a factorization over $\mathbb{Z}/p^r\mathbb{Z}$ (polynomials over that ring do not form a unique factorization domain, anyway), but approximations in $\mathbb{Q}/p^r\mathbb{Z}$ of the true factorization in $\mathbb{Q}_p[X]$.

```
? factorpadic(x<sup>2</sup> + 9, 3,5)
%1 =
[(1 + 0(3<sup>5</sup>))*x<sup>2</sup> + 0(3<sup>5</sup>)*x + (3<sup>2</sup> + 0(3<sup>5</sup>)) 1]
? factorpadic(x<sup>2</sup> + 1, 5,3)
%2 =
[ (1 + 0(5<sup>3</sup>))*x + (2 + 5 + 2*5<sup>2</sup> + 0(5<sup>3</sup>)) 1]
[(1 + 0(5<sup>3</sup>))*x + (3 + 3*5 + 2*5<sup>2</sup> + 0(5<sup>3</sup>)) 1]
```

The factors are normalized so that their leading coefficient is a power of p. The method used is a modified version of the round 4 algorithm of Zassenhaus.

If *pol* has inexact t_PADIC coefficients, this is not always well-defined; in this case, the polynomial is first made integral by dividing out the *p*-adic content, then lifted to \mathbb{Z} using truncate coefficientwise. Hence we actually factor exactly a polynomial which is only *p*-adically close to the input. To avoid pitfalls, we advise to only factor polynomials with exact rational coefficients.

ffcompomap(g)

Let k, l, m be three finite fields and f a (partial) map from l to m and g a (partial) map from k to l, return the (partial) map fog from k to m.

```
a = ffgen([3,5],'a); b = ffgen([3,10],'b); c = ffgen([3,20],'c);
m = ffembed(a, b); n = ffembed(b, c);
rm = ffinvmap(m); rn = ffinvmap(n);
nm = ffcompomap(n,m);
ffmap(n,ffmap(m,a)) == ffmap(nm, a)
%5 = 1
ffcompomap(rm, rn) == ffinvmap(nm)
%6 = 1
```

ffembed(*b*)

Given two finite fields elements a and b, return a *map* embedding the definition field of a to the definition field of b. Assume that the latter contains the former.

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? A = ffmap(m, a); ? minpoly(A) == minpoly(a) %5 = 1

ffextend(P, v)

Extend the field K of definition of a by a root of the polynomial $P \in K[X]$ assumed to be irreducible over K. Return [r, m] where r is a root of P in the extension field L and m is a map from K to L, see ffmap. If v is given, the variable name is used to display the generator of L, else the name of the variable of P is used. A generator of L can be recovered using b = ffgen(r). The image of P in L[X] can be recovered using PL = ffmap(m, P).

```
? a = ffgen([3,5],'a);
? P = x^2-a; polisirreducible(P)
%2 = 1
? [r,m] = ffextend(a, P, 'b);
? r
%3 = b^9+2*b^8+b^7+2*b^6+b^4+1
? subst(ffmap(m, P), x, r)
%4 = 0
? ffgen(r)
%5 = b
```

fffrobenius(n)

Return the n-th power of the Frobenius map over the field of definition of m.

```
? a = ffgen([3,5],'a);
? f = fffrobenius(a);
```

```
? ffmap(f,a) == a^3
%3 = 1
? g = fffrobenius(a, 5);
? ffmap(g,a) == a
%5 = 1
? h = fffrobenius(a, 2);
? h == ffcompomap(f,f)
%7 = 1
```

ffgen(v)

Return a generator for the finite field k as a t_FFELT. The field k can be given by

- its order q
- the pair [p, f] where $q = p^f$
- a monic irreducible polynomial with t_INTMOD coefficients modulo a prime.
- a t_FFELT belonging to k.

If v is given, the variable name is used to display g, else the variable of the polynomial or the t_FFELT is used, else x is used.

When only the order is specified, the function uses the polynomial generated by ffinit and is deterministic: two calls to the function with the same parameters will always give the same generator.

For efficiency, the characteristic is not checked to be prime; similarly if a polynomial is given, we do not check whether it is irreducible.

To obtain a multiplicative generator, call ffprimroot on the result.

```
? g = ffgen(16, 't);
? g.mod \\ recover the underlying polynomial.
%2 = t^4+t^3+t^2+t+1
? g.pol \\ lift g as a t_POL
%3 = t
? g.p \\ recover the characteristic
%4 = 2
? fforder(g) \\ g is not a multiplicative generator
%5 = 5
? a = ffprimroot(g) \\ recover a multiplicative generator
%6 = t^3+t^2+t
? fforder(a)
%7 = 15
```

ffinit(n, v)

Computes a monic polynomial of degree n which is irreducible over \mathbb{F}_p , where p is assumed to be prime. This function uses a fast variant of Adleman and Lenstra's algorithm.

It is useful in conjunction with ffgen; for instance if P = ffinit(3,2), you can represent elements in \mathbb{F}_{3^2} in term of g = ffgen(P, 't). This can be abbreviated as $g = ffgen(3^2, 't)$, where the defining polynomial P can be later recovered as g.mod.

ffinvmap()

m being a map from *K* to *L* two finite fields, return the partial map *p* from *L* to *K* such that for all $k \in K$, p(m(k)) = k.

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? p = ffinvmap(m); ? u = random(a); ? v = ffmap(m, u); ? ffmap(p, v^2+v+2) == u^2+u+2 %7 = 1 ? ffmap(p, b) %8 = []

fflog(g, o)

Discrete logarithm of the finite field element x in base g, i.e. an e in \mathbb{Z} such that $g^e = o$. If present, o represents the multiplicative order of g, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the order of g. It may be set as a side effect of calling ffprimroot. The result is undefined if e does not exist. This function uses

- a combination of generic discrete log algorithms (see znlog)
- a cubic sieve index calculus algorithm for large fields of degree at least 5.
- Coppersmith's algorithm for fields of characteristic at most 5.

```
? t = ffgen(ffinit(7,5));
? o = fforder(t)
%2 = 5602 \\ not a primitive root.
? fflog(t^10,t)
%3 = 10
? fflog(t^10,t, o)
%4 = 10
? g = ffprimroot(t, &o);
? o \\ order is 16806, bundled with its factorization matrix
%6 = [16806, [2, 1; 3, 1; 2801, 1]]
? fforder(g, o)
%7 = 16806
? fflog(g^10000, g, o)
%8 = 10000
```

ffmap(x)

Given a (partial) map m between two finite fields, return the image of x by m. The function is applied recursively to the component of vectors, matrices and polynomials. If m is a partial map that is not defined at x, return [].

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? P = x^2+a*x+1; ? Q = ffmap(m,P); ? ffmap(m,poldisc(P)) == poldisc(Q) %6 = 1

ffmaprel(x)

Given a (partial) map m between two finite fields, express x as an algebraic element over the codomain of m in a way which is compatible with m. The function is applied recursively to the component of vectors, matrices and polynomials.

? a = ffgen([3,5],'a); ? b = ffgen([3,10],'b); ? m = ffembed(a, b); ? mi= ffinvmap(m); ? R = ffmaprel(mi,b) %5 = Mod(b,b^2+(a+1)*b+(a^2+2*a+2))

In particular, this function can be used to compute the relative minimal polynomial, norm and trace:

? minpoly(R) %6 = x^2+(a+1)*x+(a^2+2*a+2) ? trace(R) %7 = 2*a+2 ? norm(R) %8 = a^2+2*a+2

ffnbirred(n, fl)

Computes the number of monic irreducible polynomials over \mathbb{F}_q of degree exactly n, (flag = 0 or omitted) or at most n (flag = 1).

fforder(o)

Multiplicative order of the finite field element x. If o is present, it represents a multiple of the order of the element, see DLfun (in the PARI manual); the preferred format for this parameter is [N, factor(N)], where N is the cardinality of the multiplicative group of the underlying finite field.

```
? t = ffgen(ffinit(nextprime(10^8), 5));
? g = ffprimroot(t, &o); \\ o will be useful!
? fforder(g^1000000, o)
time = 0 ms.
%5 = 5000001750000245000017150000600250008403
? fforder(g^1000000)
time = 16 ms. \\ noticeably slower, same result of course
%6 = 5000001750000245000017150000600250008403
```

ffprimroot(o)

Return a primitive root of the multiplicative group of the definition field of the finite field element x (not necessarily the same as the field generated by x). If present, o is set to a vector [ord, fa], where ord is the order of the group and fa its factorization factor(ord). This last parameter is useful in fflog and fforder, see DLfun (in the PARI manual).

```
? t = ffgen(ffinit(nextprime(10^7), 5));
? g = ffprimroot(t, &o);
? o[1]
%3 = 100000950003610006859006516052476098
? o[2]
%4 =
[2 1]
[7 2]
[31 1]
[41 1]
```

```
[67 1]
[1523 1]
[10498781 1]
[15992881 1]
[46858913131 1]
? fflog(g^1000000, g, o)
time = 1,312 ms.
%5 = 1000000
```

$\mathbf{fft}(P)$

Let $w = [1, z, ..., z^{N-1}]$ from some primitive N-roots of unity z where N is a power of 2, and P be a polynomial < N, return the unnormalized discrete Fourier transform of P, $P(w[i]), 1 \le i \le N$. Also allow P to be a vector $[p_0, ..., p_n]$ representing the polynomial $\sum p_i X^i$. Composing fft and fftinv returns N times the original input coefficients.

```
? w = rootsof1(4); fft(w, x^3+x+1)
%1 = [3, 1, -1, 1]
? fftinv(w, %)
%2 = [4, 4, 0, 4]
? Polrev(%) / 4
%3 = x^3 + x + 1
? w = powers(znprimroot(5),3); fft(w, x^3+x+1)
%4 = [Mod(3,5),Mod(1,5),Mod(4,5),Mod(1,5)]
? fftinv(w, %)
%5 = [Mod(4,5),Mod(4,5),Mod(0,5),Mod(4,5)]
```

fftinv(P)

Let $w = [1, z, ..., z^{N-1}]$ from some primitive N-roots of unity z where N is a power of 2, and P be a polynomial < N, return the unnormalized discrete Fourier transform of P, $P(1/w[i]), 1 \le i \le N$. Also allow P to be a vector $[p_0, ..., p_n]$ representing the polynomial $\sum p_i X^i$. Composing fft and fftinv returns N times the original input coefficients.

```
? w = rootsof1(4); fft(w, x^3+x+1)
%1 = [3, 1, -1, 1]
? fftinv(w, %)
%2 = [4, 4, 0, 4]
? Polrev(%) / 4
%3 = x^3 + x + 1
? N = 512; w = rootsof1(N); T = random(1000 * x^(N-1));
? U = fft(w, T);
time = 3 ms.
? V = vector(N, i, subst(T, 'x, w[i]));
time = 65 ms.
? exponent(V - U)
%7 = -97
```

```
? round(Polrev(fftinv(w,U) / N)) == T
%8 = 1
```

fileflush()

Flushes the file descriptor *n*, created via fileopen or fileextern. On files opened for writing, this function forces a write of all buffered data to the file system and completes all pending write operations. This function is implicitly called by fileclose but you may want to call it explicitly at synchronization points, for instance after writing a large result to file and before printing diagnostics on screen. (In order to be sure that the file contains the expected content on inspection.)

If n is omitted, flush all descriptors to output streams.

```
? n = fileopen("./here", "w");
? for (i = 1, 10^5, \
filewrite(n, i^2+1); \
if (i % 10000 == 0, fileflush(n)))
```

Until a fileflush or fileclose, there is no guarantee that the file contains all the expected data from previous filewrite s.

floor()

Floor of x. When x is in \mathbb{R} , the result is the largest integer smaller than or equal to x. Applied to a rational function, floor(x) returns the Euclidean quotient of the numerator by the denominator.

fold(A)

Apply the t_CLOSURE f of arity 2 to the entries of A, in order to return f(...f(f(A[1],A[2]),A[3])..., A[#A]).

```
? fold((x,y)->x*y, [1,2,3,4])
%1 = 24
? fold((x,y)->[x,y], [1,2,3,4])
%2 = [[[1, 2], 3], 4]
? fold((x,f)->f(x), [2,sqr,sqr,sqr])
%3 = 256
? fold((x,y)->(x+y)/(1-x*y),[1..5])
%4 = -9/19
? bestappr(tan(sum(i=1,5,atan(i))))
%5 = -9/19
```

frac()

Fractional part of x. Identical to x - floor(x). If x is real, the result is in [0, 1].

fromdigits(b)

Gives the integer formed by the elements of x seen as the digits of a number in base b (b = 10 by default). This is the reverse of digits:

? digits(1234,5) %1 = [1,4,4,1,4] ? fromdigits([1,4,4,1,4],5) %2 = 1234

By convention, 0 has no digits:

? fromdigits([])
%3 = 0

galoischardet(chi, o)

Let G be the group attached to the galoisinit structure gal, and let χ be the character of some representation ρ of the group G, where a polynomial variable is to be interpreted as an o-th root of 1. For instance, if [T,o] = galoischartable(gal) the characters χ are input as the columns of T.

Return the degree-1 character det ρ as the list of det $\rho(g)$, where g runs through representatives of the conjugacy classes in galoisconjclasses(gal), with the same ordering.

```
? P = x^5 - x^4 - 5*x^3 + 4*x^2 + 3*x - 1;
? polgalois(P)
%2 = [10, 1, 1, "D(5) = 5:2"]
? K = nfsplitting(P);
? gal = galoisinit(K); \\ dihedral of order 10
? [T,o] = galoischartable(gal);
? chi = T[,1]; \\ trivial character
? galoischardet(gal, chi, o)
%7 = [1, 1, 1, 1]~
? [galoischardet(gal, T[,i], o) | i <- [1..#T]] \\ all characters
%8 = [[1, 1, 1, 1]~, [1, 1, -1, 1]~, [1, 1, -1, 1]~]
```

galoischarpoly(chi, o)

Let G be the group attached to the galoisinit structure gal, and let χ be the character of some representation ρ of the group G, where a polynomial variable is to be interpreted as an o-th root of 1, e.g., if [T,o] =galoischartable(gal) and χ is a column of T. Return the list of characteristic polynomials det $(1 - \rho(g)T)$, where g runs through representatives of the conjugacy classes in galoisconjclasses(gal), with the same ordering.

```
? T = x^5 - x^4 - 5^*x^3 + 4^*x^2 + 3^*x - 1;

? polgalois(T)

%2 = [10, 1, 1, "D(5) = 5:2"]

? K = nfsplitting(T);

? gal = galoisinit(K); \\ dihedral of order 10

? [T,o] = galoischartable(gal);

? o

%5 = 5

? galoischarpoly(gal, T[,1], o) \\ T[,1] is the trivial character

%6 = [-x + 1, -x + 1, -x + 1, -x + 1]~

? galoischarpoly(gal, T[,3], o)

%7 = [x^2 - 2^*x + 1,

x^2 + (y^3 + y^2 + 1)^*x + 1,

-x^2 + 1,

x^2 + (-y^3 - y^2)^*x + 1]~
```

galoischartable()

Compute the character table of G, where G is the underlying group of the galoisinit structure *gal*. The input *gal* is also allowed to be a t_VEC of permutations that is closed under products. Let N be the number of conjugacy classes of G. Return a t_VEC [M, e] where $e \ge 1$ is an integer and M is a square t_MAT of size N giving the character table of G.

• Each column corresponds to an irreducible character; the characters are ordered by increasing dimension and the first column is the trivial character (hence contains only 1's).

• Each row corresponds to a conjugacy class; the conjugacy classes are ordered as specified by galoisconjclasses(gal), in particular the first row corresponds to the identity and gives the dimension $\chi(1)$ of the irreducible representation attached to the successive characters χ .

The value M[i, j] of the character j at the conjugacy class i is represented by a polynomial in y whose variable should be interpreted as an e-th root of unity, i.e. as the lift of

Mod(y, polcyclo(e,'y))

(Note that M is the transpose of the usual orientation for character tables.)

The integer e divides the exponent of the group G and is chosen as small as possible; for instance e = 1 when the characters are all defined over \mathbb{Q} , as is the case for S_n . Examples:

```
? K = nfsplitting(x^4+x+1);
? gal = galoisinit(K);
? [M,e] = galoischartable(gal);
? M \sim \setminus  take the transpose to get the usual orientation
%4 =
[1 \ 1 \ 1 \ 1 \ 1]
[1 - 1 - 1 1 1]
[2 \ 0 \ 0 \ -1 \ 2]
[3 -1 1 0 -1]
[3 1 - 1 0 - 1]
?е
%5 = 1
? {G = [Vecsmall([1, 2, 3, 4, 5]), Vecsmall([1, 5, 4, 3, 2]),
Vecsmall([2, 1, 5, 4, 3]), Vecsmall([2, 3, 4, 5, 1]),
Vecsmall([3, 2, 1, 5, 4]), Vecsmall([3, 4, 5, 1, 2]),
Vecsmall([4, 3, 2, 1, 5]), Vecsmall([4, 5, 1, 2, 3]),
Vecsmall([5, 1, 2, 3, 4]), Vecsmall([5, 4, 3, 2, 1])];}
\backslash G = D10
? [M,e] = galoischartable(G);
? M~
%8 =
[1 \ 1 \ 1 \ 1]
[1 - 1 1 1]
[2 \ 0 \ -y^3 \ -y^2 \ -1 \ y^3 \ +y^2]
[2 \ 0 \ y^3 + y^2 - y^3 - y^2 - 1]
?е
%9 = 5
```

galoisconjclasses()

gal being output by galoisinit, return the list of conjugacy classes of the underlying group. The ordering of the classes is consistent with galoischartable and the trivial class comes first.

```
? G = galoisinit(x^6+108);
? galoisidentify(G)
```

```
%2 = [6, 1] \\ S_3
? S = galoisconjclasses(G)
%3 = [[Vecsmall([1,2,3,4,5,6])],
  [Vecsmall([3,1,2,6,4,5]),Vecsmall([2,3,1,5,6,4])],
  [Vecsmall([6,5,4,3,2,1]),Vecsmall([5,4,6,2,1,3]),
  Vecsmall([4,6,5,1,3,2])]]
? [[permorder(c[1]),#c] | c <- S ]
%4 = [[1,1], [3,2], [2,3]]</pre>
```

This command also accepts subgroups returned by galoissubgroups:

```
? subs = galoissubgroups(G); H = subs[5];
? galoisidentify(H)
%2 = [2, 1] \\ Z/2
? S = galoisconjclasses(subgroups_of_G[5]);
? [[permorder(c[1]),#c] | c <- S ]
%4 = [[1,1], [2,1]]
```

galoisexport(flag)

gal being be a Galois group as output by galoisinit, export the underlying permutation group as a string suitable for (no flags or flag = 0) GAP or (flag = 1) Magma. The following example compute the index of the underlying abstract group in the GAP library:

```
? G = galoisinit(x^6+108);
? s = galoisexport(G)
%2 = "Group((1, 2, 3)(4, 5, 6), (1, 4)(2, 6)(3, 5))"
? extern("echo \"IdGroup("s");\" | gap -q")
%3 = [6, 1]
? galoisidentify(G)
%4 = [6, 1]
```

This command also accepts subgroups returned by galoissubgroups.

To *import* a GAP permutation into gp (for galoissubfields for instance), the following GAP function may be useful:

```
PermToGP := function(p, n)
return Permuted([1..n],p);
end;
gap> p:= (1,26)(2,5)(3,17)(4,32)(6,9)(7,11)(8,24)(10,13)(12,15)(14,27)
  (16,22)(18,28)(19,20)(21,29)(23,31)(25,30)
gap> PermToGP(p,32);
[ 26, 5, 17, 32, 2, 9, 11, 24, 6, 13, 7, 15, 10, 27, 12, 22, 3, 28, 20, 19,
  29, 16, 31, 8, 30, 1, 14, 18, 21, 25, 23, 4 ]
```

galoisfixedfield(perm, flag, v)

gal being be a Galois group as output by galoisinit and perm an element of gal.group, a vector of such elements or a subgroup of gal as returned by galoissubgroups, computes the fixed field of gal by the automorphism defined by the permutations perm of the roots gal.roots. P is guaranteed to be squarefree modulo gal.p.

If no flags or flag = 0, output format is the same as for nfsubfield, returning [P, x] such that P is a polynomial defining the fixed field, and x is a root of P expressed as a polmod in *gal.pol*.

If flag = 1 return only the polynomial P.

If flag = 2 return [P, x, F] where P and x are as above and F is the factorization of gal.pol over the field defined by P, where variable v (y by default) stands for a root of P. The priority of v must be less than the priority of the variable of gal.pol (see priority (in the PARI manual)). In this case, P is also expressed in the variable v for compatibility with F. Example:

? G = galoisinit(x⁴+1); ? galoisfixedfield(G,G.group[2],2) %2 = [y² - 2, Mod(- x³ + x, x⁴ + 1), [x² - y*x + 1, x² + y*x + 1]]

computes the factorization $x^4 + 1 = (x^2 - \sqrt{2}x + 1)(x^2 + \sqrt{2}x + 1)$

galoisidentify()

gal being be a Galois group as output by galoisinit, output the isomorphism class of the underlying abstract group as a two-components vector [o, i], where o is the group order, and i is the group index in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien.

This command also accepts subgroups returned by galoissubgroups.

The current implementation is limited to degree less or equal to 127. Some larger "easy" orders are also supported.

The output is similar to the output of the function IdGroup in GAP4. Note that GAP4 IdGroup handles all groups of order less than 2000 except 1024, so you can use galoisexport and GAP4 to identify large Galois groups.

galoisinit(den)

Computes the Galois group and all necessary information for computing the fixed fields of the Galois extension K/\mathbb{Q} where K is the number field defined by *pol* (monic irreducible polynomial in $\mathbb{Z}[X]$ or a number field as output by nfinit). The extension K/\mathbb{Q} must be Galois with Galois group "weakly" super-solvable, see below; returns 0 otherwise. Hence this permits to quickly check whether a polynomial of order strictly less than 48 is Galois or not.

The algorithm used is an improved version of the paper "An efficient algorithm for the computation of Galois automorphisms", Bill Allombert, Math. Comp, vol. 73, 245, 2001, pp. 359–375.

A group G is said to be "weakly" super-solvable if there exists a normal series

 $1=H_0 \triangleleft H_1 \triangleleft \ldots \triangleleft H_{n-1} \triangleleft H_n$

such that each H_i is normal in G and for i < n, each quotient group H_{i+1}/H_i is cyclic, and either $H_n = G$ (then G is super-solvable) or G/H_n is isomorphic to either A_4 , S_4 or the group (3x3) : 4 (GAP4(36,9)) then $[o_1, ..., o_q]$ ends by [3, 3, 4].

In practice, almost all small groups are WKSS, the exceptions having order 48(2), 56(1), 60(1), 72(3), 75(1), 80(1), 96(10), 112(1), 120(3) and >= 144.

This function is a prerequisite for most of the galoisxxx routines. For instance:

```
P = x^6 + 108;
G = galoisinit(P);
L = galoissubgroups(G);
vector(#L, i, galoisisabelian(L[i],1))
vector(#L, i, galoisidentify(L[i]))
```

The output is an 8-component vector gal.

gal[1] contains the polynomial pol (:emphasis:`gal.pol`).

gal[2] is a three-components vector [p, e, q] where p is a prime number (:emphasis:`gal.p`) such that pol totally split modulo p, e is an integer and $q = p^e$ (:emphasis:`gal.mod`) is the modulus of the roots in :emphasis:`gal.roots`.

gal[3] is a vector L containing the p-adic roots of pol as integers implicitly modulo :emphasis:`gal.mod`. (:emphasis:`gal.roots`).

gal[4] is the inverse of the Vandermonde matrix of the *p*-adic roots of *pol*, multiplied by gal[5].

gal[5] is a multiple of the least common denominator of the automorphisms expressed as polynomial in a root of pol.

gal[6] is the Galois group G expressed as a vector of permutations of L (:emphasis:`gal.group`).

gal[7] is a generating subset $S = [s_1, ..., s_g]$ of G expressed as a vector of permutations of L (:emphasis:`gal.gen`).

gal[8] contains the relative orders $[o_1, ..., o_g]$ of the generators of S (:emphasis:`gal.orders`).

Let H_n be as above, we have the following properties:

* if $G/H_n A_4$ then $[o_1, ..., o_q]$ ends by [2, 2, 3].

* if $G/H_n S_4$ then $[o_1, ..., o_q]$ ends by [2, 2, 3, 2].

* if $G/H_n(3x3): 4$ (GAP4(36,9)) then $[o_1, ..., o_q]$ ends by [3, 3, 4].

* for $1 \le i \le g$ the subgroup of G generated by $[s_1, ..., s_i]$ is normal, with the exception of i = g - 2 in the A_4 case and of i = g - 3 in the S_4 case.

* the relative order o_i of s_i is its order in the quotient group $G/\langle s_1, ..., s_{i-1} \rangle$, with the same exceptions.

* for any $x \in G$ there exists a unique family $[e_1, ..., e_g]$ such that (no exceptions):

- for $1 \le i \le g$ we have $0 \le e_i < o_i$

$$-x = g_1^{e_1} g_2^{e_2} \dots g_n^{e_n}$$

If present den must be a suitable value for gal[5].

galoisisabelian(flag)

gal being as output by galoisinit, return 0 if gal is not an abelian group, and the HNF matrix of gal over gal.gen if flag = 0, 1 if flag = 1, and the SNF matrix of gal if flag = 2.

This command also accepts subgroups returned by galoissubgroups.

galoisisnormal(subgrp)

gal being as output by galoisinit, and subgrp a subgroup of gal as output by galoissubgroups, return 1 if subgrp is a normal subgroup of gal, else return 0.

This command also accepts subgroups returned by galoissubgroups.

galoispermtopol(perm)

gal being a Galois group as output by galoisinit and *perm* a element of *gal.group*, return the polynomial defining the Galois automorphism, as output by nfgaloisconj, attached to the permutation *perm* of the roots *gal.roots. perm* can also be a vector or matrix, in this case, galoispermtopol is applied to all components recursively.

Note that

```
G = galoisinit(pol);
galoispermtopol(G, G[6])~
```

is equivalent to nfgaloisconj(pol), if degree of pol is greater or equal to 2.

galoissubcyclo(H, fl, v)

Computes the subextension of $\mathbb{Q}(\zeta_n)$ fixed by the subgroup $H \subset (\mathbb{Z}/n\mathbb{Z})^*$. By the Kronecker-Weber theorem, all abelian number fields can be generated in this way (uniquely if n is taken to be minimal).

The pair (n, H) is deduced from the parameters (N, H) as follows

- N an integer: then n = N; H is a generator, i.e. an integer or an integer modulo n; or a vector of generators.
- N the output of znstar(n) or znstar(n, 1). H as in the first case above, or a matrix, taken to be a HNF left divisor of the SNF for (Z/nZ)* (:math:`N.cyc`), giving the generators of H in terms of :math:`N.gen`.
- N the output of bnrinit(bnfinit(y), :math:`m)` where m is a module. H as in the first case, or a matrix taken to be a HNF left divisor of the SNF for the ray class group modulo m (of type :math:`N.cyc`), giving the generators of H in terms of :math:`N.bid.gen` (= :math:`N`.gen if N includes generators).

In this last case, beware that H is understood relatively to N; in particular, if the infinite place does not divide the module, e.g if m is an integer, then it is not a subgroup of $(\mathbb{Z}/n\mathbb{Z})^*$, but of its quotient by 1.

If fl = 0, compute a polynomial (in the variable v) defining the subfield of $\mathbb{Q}(\zeta_n)$ fixed by the subgroup H of $(\mathbb{Z}/n\mathbb{Z})^*$.

If fl = 1, compute only the conductor of the abelian extension, as a module.

If fl = 2, output [pol, N], where pol is the polynomial as output when fl = 0 and N the conductor as output when fl = 1.

The following function can be used to compute all subfields of $\mathbb{Q}(\zeta_n)$ (of exact degree d, if d is set):

```
subcyclo(n, d = -1)=
{ my(bnr,L,IndexBound);
IndexBound = if (d < 0, n, [d]);
bnr = bnrinit(bnfinit(y), [n,[1]]);
L = subgrouplist(bnr, IndexBound, 1);
vector(#L,i, galoissubcyclo(bnr,L[i]));
}</pre>
```

Setting L = subgrouplist(bnr, IndexBound) would produce subfields of exact conductor noo.

galoissubfields(flag, v)

Outputs all the subfields of the Galois group G, as a vector. This works by applying galoisfixedfield to all subgroups. The meaning of *flag* is the same as for galoisfixedfield.

galoissubgroups()

Outputs all the subgroups of the Galois group gal. A subgroup is a vector [*gen*, *orders*], with the same meaning as for *gal.gen* and *gal.orders*. Hence *gen* is a vector of permutations generating the subgroup, and *orders* is the relatives orders of the generators. The cardinality of a subgroup is the product of the relative orders. Such subgroup can be used instead of a Galois group in the following command: galoisisabelian, galoissubgroups, galoisexport and galoisidentify.

To get the subfield fixed by a subgroup sub of gal, use

```
galoisfixedfield(gal,sub[1])
```

gamma (precision)

For s a complex number, evaluates Euler's gamma function

$$\Gamma(s) = \int_0^o ot^{s-1} \exp(-t) dt.$$

Error if s is a nonpositive integer, where Γ has a pole.

For s a t_PADIC, evaluates the Morita gamma function at s, that is the unique continuous p-adic function on the p-adic integers extending $\Gamma_p(k) = (-1)^k \prod_{i \le k}^{\prime} j$, where the prime means that p does not divide j.

```
? gamma(1/4 + 0(5^10))
%1= 1 + 4*5 + 3*5^4 + 5^6 + 5^7 + 4*5^9 + 0(5^10)
? algdep(%,4)
%2 = x^4 + 4*x^2 + 5
```

gammah(precision)

Gamma function evaluated at the argument x + 1/2.

gammamellininv(t, m, precision)

Returns the value at t of the inverse Mellin transform G initialized by gammamellininvinit. If the optional parameter m is present, return the m-th derivative $G^{(m)}(t)$.

```
? G = gammamellininvinit([0]);
? gammamellininv(G, 2) - 2*exp(-Pi*2*2)
%2 = -4.484155085839414627 E-44
```

The shortcut

gammamellininv(A,t,m)

for

```
gammamellininv(gammamellininvinit(A,m), t)
```

is available.

gammamellininvasymp(serprec, n)

Return the first n terms of the asymptotic expansion at infinity of the m-th derivative $K^{(m)}(t)$ of the inverse Mellin transform of the function

$$f(s) = \Gamma_{\mathbb{R}}(s+a_1)...\Gamma_{\mathbb{R}}(s+a_d),$$

where A is the vector $[a_1, ..., a_d]$ and $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2}\Gamma(s/2)$ (Euler's gamma). The result is a vector [M[1]...M[n]] with M[1] = 1, such that

$$K^{(m)}(t) = \sqrt{2^{d+1}/dt^{a+m(2/d-1)}} e^{-d\pi t^{2/d}} \sum_{n>=0} M[n+1](\pi t^{2/d})^{-n}$$

with $a = (1 - d + \sum_{1 \le j \le d} a_j)/d$. We also allow A to be the output of gammamellininvinit.

gammamellininvinit(m, precision)

Initialize data for the computation by gammamellininv of the m-th derivative of the inverse Mellin transform of the function

$$f(s) = \Gamma_{\mathbb{R}}(s+a_1)...\Gamma_{\mathbb{R}}(s+a_d)$$

where A is the vector $[a_1, ..., a_d]$ and $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2}\Gamma(s/2)$ (Euler's gamma). This is the special case of Meijer's G functions used to compute L-values via the approximate functional equation. By extension, A is allowed to be an Ldata or an Linit, understood as the inverse Mellin transform of the L-function gamma factor.

Caveat. Contrary to the PARI convention, this function guarantees an *absolute* (rather than relative) error bound.

For instance, the inverse Mellin transform of $\Gamma_{\mathbb{R}}(s)$ is $2 \exp(-\pi z^2)$:

? G = gammamellininvinit([0]); ? gammamellininv(G, 2) - 2*exp(-Pi*2^2) %2 = -4.484155085839414627 E-44

The inverse Mellin transform of $\Gamma_{\mathbb{R}}(s+1)$ is $2z \exp(-\pi z^2)$, and its second derivative is $4\pi z \exp(-\pi z^2)(2\pi z^2 - 3)$:

? G = gammamellininvinit([1], 2); ? a(z) = 4*Pi*z*exp(-Pi*z^2)*(2*Pi*z^2-3); ? b(z) = gammamellininv(G,z); ? t(z) = b(z) - a(z); ? t(3/2) %3 = -1.4693679385278593850 E-39

gcd(*y*)

Creates the greatest common divisor of x and y. If you also need the u and v such that x * u + y * v = gcd(x, y), use the gcdext function. x and y can have rather quite general types, for instance both rational numbers. If y is omitted and x is a vector, returns the gcd of all components of x, i.e. this is equivalent to content(x).

When x and y are both given and one of them is a vector/matrix type, the GCD is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to gcd(x, y[i]), resp. gcd(x, y[,i]). Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, gcd is not commutative.

The algorithm used is a naive Euclid except for the following inputs:

- integers: use modified right-shift binary ("plus-minus" variant).
- univariate polynomials with coefficients in the same number field (in particular rational): use modular gcd algorithm.
- general polynomials: use the subresultant algorithm if coefficient explosion is likely (non modular coefficients).

If u and v are polynomials in the same variable with *inexact* coefficients, their gcd is defined to be scalar, so that

? a = x + 0.0; gcd(a,a) %1 = 1 ? b = y*x + 0(y); gcd(b,b) %2 = y ? c = 4*x + 0(2^3); gcd(c,c) %3 = 4

A good quantitative check to decide whether such a gcd "should be" nontrivial, is to use polresultant: a value close to 0 means that a small deformation of the inputs has nontrivial gcd. You may also use gcdext, which does try to compute an approximate gcd d and provides u, v to check whether ux + vy is close to d.

gcdext(y)

Returns [u, v, d] such that d is the gcd of x, y, x * u + y * v = gcd(x, y), and u and v minimal in a natural sense. The arguments must be integers or polynomials.

? [u, v, d] = gcdext(32,102) %1 = [16, -5, 2] ? d %2 = 2 ? gcdext(x^2-x, x^2+x-2) %3 = [-1/2, 1/2, x - 1] If x, y are polynomials in the same variable and *inexact* coefficients, then compute u, v, d such that x * u + y * v = d, where d approximately divides both and x and y; in particular, we do not obtain gcd(x,y) which is *defined* to be a scalar in this case:

? a = x + 0.0; gcd(a,a) %1 = 1 ? gcdext(a,a) %2 = [0, 1, x + 0.E-28] ? gcdext(x-Pi, 6*x^2-zeta(2)) %3 = [-6*x - 18.8495559, 1, 57.5726923]

For inexact inputs, the output is thus not well defined mathematically, but you obtain explicit polynomials to check whether the approximation is close enough for your needs.

genus2red(p)

Let PQ be a polynomial P, resp. a vector [P, Q] of polynomials, with rational coefficients. Determines the reduction at p > 2 of the (proper, smooth) genus 2 curve C/\mathbb{Q} , defined by the hyperelliptic equation $y^2 = P(x)$, resp. $y^2 + Q(x) * y = P(x)$. (The special fiber X_p of the minimal regular model X of C over \mathbb{Z} .)

If p is omitted, determines the reduction type for all (odd) prime divisors of the discriminant.

This function was rewritten from an implementation of Liu's algorithm by Cohen and Liu (1994), genus2reduction-0.3, see http://www.math.u-bordeaux.fr/~liu/G2R/.

CAVEAT. The function interface may change: for the time being, it returns [N, FaN, T, V] where N is either the local conductor at p or the global conductor, FaN is its factorization, $y^2 = T$ defines a minimal model over $\mathbb{Z}[1/2]$ and V describes the reduction type at the various considered p. Unfortunately, the program is not complete for p = 2, and we may return the odd part of the conductor only: this is the case if the factorization includes the (impossible) term 2^{-1} ; if the factorization contains another power of 2, then this is the exact local conductor at 2 and N is the global conductor.

```
? default(debuglevel, 1);
? genus2red(x^6 + 3 x^3 + 63, 3)
(potential) stable reduction: [1, []]
reduction at p: [III{9}] page 184, [3, 3], f = 10
%1 = [59049, Mat([3, 10]), x^6 + 3*x^3 + 63, [3, [1, []],
 ["[III{9}] page 184", [3, 3]]]]
? [N, FaN, T, V] = genus2red(x^3-x^2-1, x^2-x); \\ X_1(13), global reduction
p = 13
(potential) stable reduction: [5, [Mod(0, 13), Mod(0, 13)]]
reduction at p: [I{0}-II-0] page 159, [], f = 2
? N
%3 = 169
? FaN
\%4 = Mat([13, 2]) \setminus in particular, good reduction at 2 !
? T
5 = x^{6} + 58^{*}x^{5} + 1401^{*}x^{4} + 18038^{*}x^{3} + 130546^{*}x^{2} + 503516^{*}x + 808561^{*}
? V
%6 = [[13, [5, [Mod(0, 13), Mod(0, 13)]], ["[I{0}-II-0] page 159", []]]]
```

We now first describe the format of the vector $V = V_p$ in the case where p was specified (local reduction at p): it is a triple [p, stable, red]. The component stable = [type, vecj] contains information about the stable reduction after a field extension; depending on type s, the stable reduction is

• 1: smooth (i.e. the curve has potentially good reduction). The Jacobian J(C) has potentially good reduction.

- 2: an elliptic curve E with an ordinary double point; *vecj* contains $j \mod p$, the modular invariant of E. The (potential) semi-abelian reduction of J(C) is the extension of an elliptic curve (with modular invariant $j \mod p$) by a torus.
- 3: a projective line with two ordinary double points. The Jacobian J(C) has potentially multiplicative reduction.
- 4: the union of two projective lines crossing transversally at three points. The Jacobian J(C) has potentially multiplicative reduction.
- 5: the union of two elliptic curves E₁ and E₂ intersecting transversally at one point; vecj contains their modular invariants j₁ and j₂, which may live in a quadratic extension of F_p and need not be distinct. The Jacobian J(C) has potentially good reduction, isomorphic to the product of the reductions of E₁ and E₂.
- 6: the union of an elliptic curve *E* and a projective line which has an ordinary double point, and these two components intersect transversally at one point; *vecj* contains *j* mod *p*, the modular invariant of *E*. The (potential) semi-abelian reduction of *J*(*C*) is the extension of an elliptic curve (with modular invariant *j* mod *p*) by a torus.
- 7: as in type 6, but the two components are both singular. The Jacobian J(C) has potentially multiplicative reduction.

The component red = [NUtype, neron] contains two data concerning the reduction at p without any ramified field extension.

The *NUtype* is a t_STR describing the reduction at p of C, following Namikawa-Ueno, *The complete classification of fibers in pencils of curves of genus two*, Manuscripta Math., vol. 9, (1973), pages 143-186. The reduction symbol is followed by the corresponding page number or page range in this article.

The second datum *neron* is the group of connected components (over an algebraic closure of \mathbb{F}_p) of the Néron model of J(C), given as a finite abelian group (vector of elementary divisors).

If p = 2, the *red* component may be omitted altogether (and replaced by [], in the case where the program could not compute it. When p was not specified, V is the vector of all V_p , for all considered p.

Notes about Namikawa-Ueno types.

- A lower index is denoted between braces: for instance, [I{2}-II-5] means [I_2-II-5].
- If K and K' are Kodaira symbols for singular fibers of elliptic curves, then [:math:`K-K'-m]` and [:math:`K'-K-m]` are the same.

We define a total ordering on Kodaira symbol by fixing I < I* < II < II*, If the reduction type is the same, we order by the number of components, e.g. $I_2 < I_4$, etc. Then we normalize our output so that K <= K'.

- [:math: K-K'--1] is [:math: $K-K'-\alpha$] in the notation of Namikawa-Ueno.
- The figure [21_0-m] in Namikawa-Ueno, page 159, must be denoted by [21_0-(m+1)].

halfgcd(y)

Let inputs x and y be both integers, or both polynomials in the same variable. Return a vector [M, [a,b]~], where M is an invertible $2x^2$ matrix such that $M^*[x,y] \sim = [a,b] \sim$, where b is small. More precisely,

• polynomial case: $\det M$ has degree 0 and we have

 $\deg a \ge ceil\max(\deg x, \deg y))/2 > \deg b.$

• integer case: $\det M = 1$ and we have

 $Assuming: math: `x`and: math: `y`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and: math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative, then: math: `M^{-1}`has nonnegative coefficients, and ``math: `Q`are nonnegative$

hammingweight()

If x is a t_INT, return the binary Hamming weight of ||x||. Otherwise x must be of type t_POL, t_VEC, t_COL, t_VECSMALL, or t_MAT and the function returns the number of nonzero coefficients of x.

? hammingweight(15) %1 = 4 ? hammingweight(x^100 + 2*x + 1) %2 = 3 ? hammingweight([Mod(1,2), 2, Mod(0,3)]) %3 = 2 ? hammingweight(matid(100)) %4 = 100

hilbert(y, p)

Hilbert symbol of x and y modulo the prime p, p = 0 meaning the place at infinity (the result is undefined if p! = 0 is not prime).

It is possible to omit p, in which case we take p = 0 if both x and y are rational, or one of them is a real number. And take p = q if one of x, y is a t_INTMOD modulo q or a q-adic. (Incompatible types will raise an error.)

hyperellcharpoly()

X being a nonsingular hyperelliptic curve defined over a finite field, return the characteristic polynomial of the Frobenius automorphism. X can be given either by a squarefree polynomial P such that $X : y^2 = P(x)$ or by a vector [P, Q] such that $X : y^2 + Q(x)y = P(x)$ and $Q^2 + 4P$ is squarefree.

hyperellpadicfrobenius(q, n)

Let X be the curve defined by $y^2 = Q(x)$, where Q is a polynomial of degree d over \mathbb{Q} and $q \ge d$ is a prime such that X has good reduction at q. Return the matrix of the Frobenius endomorphism φ on the crystalline module $D_p(X) = \mathbb{Q}_p \otimes H^1_{dR}(X/\mathbb{Q})$ with respect to the basis of the given model $(\omega, x\omega, ..., x^{g-1}\omega)$, where $\omega = dx/(2y)$ is the invariant differential, where g is the genus of X (either d = 2g + 1 or d = 2g + 2). The characteristic polynomial of φ is the numerator of the zeta-function of the reduction of the curve X modulo q. The matrix is computed to absolute q-adic precision q^n .

Alternatively, q may be of the form [T, p] where p is a prime, T is a polynomial with integral coefficients whose projection to $\mathbb{F}_p[t]$ is irreducible, X is defined over $K = \mathbb{Q}[t]/(T)$ and has good reduction to the finite field $\mathbb{F}_q = \mathbb{F}_p[t]/(T)$. The matrix of φ on $D_q(X) = \mathbb{Q}_q \otimes H^1_{dR}(X/K)$ is computed to absolute p-adic precision p^n .

```
? M=hyperellpadicfrobenius(x^5+'a*x+1,['a^2+1,3],10);
? liftall(M)
[48107*a + 38874 9222*a + 54290 41941*a + 8931 39672*a + 28651]
[ 21458*a + 4763 3652*a + 22205 31111*a + 42559 39834*a + 40207]
[ 13329*a + 4140 45270*a + 25803 1377*a + 32931 55980*a + 21267]
[15086*a + 26714 33424*a + 4898 41830*a + 48013 5913*a + 24088]
? centerlift(simplify(liftpol(charpoly(M))))
%8 = x^4+4*x^2+81
? hyperellcharpoly((x^5+Mod(a,a^2+1)*x+1)*Mod(1,3))
%9 = x^4+4*x^2+81
```

hyperellratpoints(h, flag)

X being a nonsingular hyperelliptic curve given by an rational model, return a vector containing the affine rational points on the curve of naive height less than h.a If flag = 1, stop as soon as a point is found; return either an empty vector or a vector containing a single point.

X is given either by a squarefree polynomial P such that $X : y^2 = P(x)$ or by a vector [P,Q] such that $X : y^2 + Q(x)y = P(x)$ and $Q^2 + 4P$ is squarefree.

The parameter h can be

- an integer H: find the points [n/d, y] whose abscissas x = n/d have naive height (= max(||n||, d)) less than H;
- a vector [N, D] with $D \le N$: find the points [n/d, y] with $||n|| \le N$, $d \le D$.
- a vector $[N, [D_1, D_2]]$ with $D_1 < D_2 <= N$ find the points [n/d, y] with $||n|| \le N$ and $D_1 \le d \le D_2$.

hypergeom(*D*, *z*, *precision*)

General hypergeometric function, where N and D are the vector of parameters in the numerator and denominator respectively, evaluated at the complex argument z.

This function implements hypergeometric functions

$${}_{p}F_{q}((a_{i})_{1 < =i < =p}, (b_{j})_{1 < =j < =q}; z) = \sum_{n > =0} (\prod_{1 < =i < =p} (a_{i})_{n}) / (\prod_{1 < =j < =q} (b_{j})_{n})(z^{n}) / (n!),$$

where $(a)_n = a(a+1)...(a+n-1)$ is the rising Pochammer symbol. For this to make sense, none of the b_j must be a negative or zero integer. The corresponding general GP command is

hypergeom([a1,a2,...,ap], [b1,b2,...,bq], z)

Whenever p = 1 or q = 1, a one-element vector can be replaced by the element it contains. Whenever p = 0 or q = 0, an empty vector can be omitted. For instance hypergeom(,b,z) computes ${}_{0}F_{1}(;b;z)$.

We distinguish three kinds of such functions according to their radius of convergence R:

- q >= p: R = oo.
- q = p 1: R = 1. Nonetheless, by integral representations, ${}_{p}F_{q}$ can be analytically continued outside the disc of convergence.
- $q \le p-2$: R = 0. By integral representations, one can make sense of the function in a suitable domain.

The list of implemented functions and their domain of validity in our implementation is as follows:

F01: hypergeom(, a, z) (or [a]). This is essentially a Bessel function and computed as such. R = oo.

F10: hypergeom(a,,z) This is $(1-z)^{-a}$.

F11: hypergeom(a,b,z) is the Kummer confluent hypergeometric function, computed by summing the series. R = oo

F20: hypergeom([a,b],,z). R = 0, computed as

$$(1)/(\Gamma(a)) \int_0^o ot^{a-1} (1-zt)^{-b} e^{-t} dt.$$

F21: hypergeom([a,b],c,z) (or [c]). R = 1, extended by

$$(\Gamma(c))/(\Gamma(b)\Gamma(c-b))\int_0^1 t^{b-1}(1-t)^{c-b-1}(1-zt)^a dt.$$

This is Gauss's Hypergeometric function, and almost all of the implementation work is done for this function. F31: hypergeom([a,b,c],d,z) (or [d]). R = 0, computed as

$$(1)/(\Gamma(a))\int_0^o ot^{a-1}e_2^{-t}F_1(b,c;d;tz)dt$$

F32: hypergeom([a,b,c],[d,e],z). R = 1, extended by

$$(\Gamma(e))/(\Gamma(c)\Gamma(e-c))\int_0^1 t^{c-1}(1-t)_2^{e-c-1}F_1(a,b;d;tz)dt.$$

For other inputs: if R = oo or R = 1 and $||z|| < 1 - \varepsilon$ is not too close to the circle of convergence, we simply sum the series.

This identity is due to Bercu.

hyperu(b, z, precision)

U-confluent hypergeometric function with complex parameters a, b, z. Note that ${}_2F_0(a, b, z) = (-z)^{-a}U(a, a + 1 - b, -1/z)$,

? hyperu(1, 3/2, I) %1 = 0.23219... - 0.80952...*I ? -I * hypergeom([1, 1+1-3/2], [], -1/I) %2 = 0.23219... - 0.80952...*I

idealadd(x, y)

Sum of the two ideals x and y in the number field nf. The result is given in HNF.

```
? K = nfinit(x^2 + 1);
? a = idealadd(K, 2, x + 1) \\ ideal generated by 2 and 1+I
%2 =
[2 1]
[0 1]
? pr = idealprimedec(K, 5)[1]; \\ a prime ideal above 5
? idealadd(K, a, pr) \\ coprime, as expected
%4 =
[1 0]
[0 1]
```

This function cannot be used to add arbitrary Z-modules, since it assumes that its arguments are ideals:

```
? b = Mat([1,0]~);
? idealadd(K, b, b) \\ only square t_MATs represent ideals
*** idealadd: nonsquare t_MAT in idealtyp.
? c = [2, 0; 2, 0]; idealadd(K, c, c) \\ nonsense
%6 =
[2 0]
```

```
[0 2]
? d = [1, 0; 0, 2]; idealadd(K, d, d) \\ nonsense
%7 =
[1 0]
[0 1]
```

In the last two examples, we get wrong results since the matrices c and d do not correspond to an ideal: the \mathbb{Z} -span of their columns (as usual interpreted as coordinates with respect to the integer basis K.zk) is not an O_K -module. To add arbitrary \mathbb{Z} -modules generated by the columns of matrices A and B, use mathnf(concat(A,B)).

idealaddtoone(x, y)

x and y being two co-prime integral ideals (given in any form), this gives a two-component row vector [a, b] such that $a \in x, b \in y$ and a + b = 1.

The alternative syntax idealaddtoone(nf, v), is supported, where v is a k-component vector of ideals (given in any form) which sum to \mathbb{Z}_K . This outputs a k-component vector e such that $e[i] \in x[i]$ for $1 \le i \le k$ and $\sum_{1 \le i \le k} e[i] = 1$.

idealappr(x, flag)

If x is a fractional ideal (given in any form), gives an element α in nf such that for all prime ideals p such that the valuation of x at p is nonzero, we have $v_p(\alpha) = v_p(x)$, and $v_p(\alpha) \ge 0$ for all other p.

The argument x may also be given as a prime ideal factorization, as output by idealfactor, but allowing zero exponents. This yields an element α such that for all prime ideals p occurring in x, $v_p(\alpha) = v_p(x)$; for all other prime ideals, $v_p(\alpha) \ge 0$.

flag is deprecated (ignored), kept for backward compatibility.

idealchinese(x, y)

x being a prime ideal factorization (i.e. a 2-columns matrix whose first column contains prime ideals and the second column contains integral exponents), y a vector of elements in nf indexed by the ideals in x, computes an element b such that

 $v_p(b-y_p) \ge v_p(x)$ for all prime ideals in x and $v_p(b) \ge 0$ for all other p.

```
? K = nfinit(t<sup>2</sup>-2);
? x = idealfactor(K, 2<sup>2</sup>*3)
%2 =
[[2, [0, 1]~, 2, 1, [0, 2; 1, 0]] 4]
[ [3, [3, 0]~, 1, 2, 1] 1]
? y = [t,1];
? idealchinese(K, x, y)
%4 = [4, -3]~
```

The argument x may also be of the form [x, s] where the first component is as above and s is a vector of signs, with r_1 components s_i in -1, 0, 1: if σ_i denotes the *i*-th real embedding of the number field, the element b returned satisfies further $sign(\sigma_i(b)) = s_i$ for all *i* such that $s_i = 1$. In other words, the sign is fixed to s_i at the *i*-th embedding whenever s_i is nonzero.

```
? idealchinese(K, [x, [1,1]], y)
%5 = [16, -3]~
? idealchinese(K, [x, [-1,-1]], y)
```

%6 = [-20, -3]~ ? idealchinese(K, [x, [1,-1]], y) %7 = [4, -3]~

If y is omitted, return a data structure which can be used in place of x in later calls and allows to solve many chinese remainder problems for a given x more efficiently.

```
? C = idealchinese(K, [x, [1,1]]);
? idealchinese(K, C, y) \\ as above
%9 = [16, -3]~
? for(i=1,10^4, idealchinese(K,C,y)) \\ ... but faster !
time = 80 ms.
? for(i=1,10^4, idealchinese(K,[x,[1,1]],y))
time = 224 ms.
```

Finally, this structure is itself allowed in place of x, the new s overriding the one already present in the structure. This allows to initialize for different sign conditions more efficiently when the underlying ideal factorization remains the same.

```
? D = idealchinese(K, [C, [1,-1]]); \\ replaces [1,1]
? idealchinese(K, D, y)
%13 = [4, -3]~
? for(i=1,10^4,idealchinese(K,[C,[1,-1]]))
time = 40 ms. \\ faster than starting from scratch
? for(i=1,10^4,idealchinese(K,[x,[1,-1]]))
time = 128 ms.
```

idealcoprime(x, y)

Given two integral ideals x and y in the number field nf, returns a β in the field, such that β .x is an integral ideal coprime to y.

idealdiv(x, y, flag)

Quotient $x \cdot y^{-1}$ of the two ideals x and y in the number field nf. The result is given in HNF.

If *flag* is nonzero, the quotient $x.y^{-1}$ is assumed to be an integral ideal. This can be much faster when the norm of the quotient is small even though the norms of x and y are large. More precisely, the algorithm cheaply removes all maximal ideals above rational primes such that $v_p(Nx) = v_p(Ny)$.

idealdown(x)

Let nf be a number field as output by nfinit, and x a fractional ideal. This function returns the nonnegative rational generator of $x \cap \mathbb{Q}$. If x is an extended ideal, the extended part is ignored.

```
? nf = nfinit(y^2+1);
? idealdown(nf, -1/2)
%2 = 1/2
? idealdown(nf, (y+1)/3)
%3 = 2/3
? idealdown(nf, [2, 11]~)
%4 = 125
? x = idealprimedec(nf, 2)[1]; idealdown(nf, x)
%5 = 2
? idealdown(nf, [130, 94; 0, 2])
%6 = 130
```

idealfactor(x, lim)

Factors into prime ideal powers the ideal x in the number field nf. The output format is similar to the factor function, and the prime ideals are represented in the form output by the idealprimedec function. If *lim* is set, return partial factorization, including only prime ideals above rational primes < lim.

```
? nf = nfinit(x^3-2);
? idealfactor(nf, x) \\ a prime ideal above 2
%2 =
[[2, [0, 1, 0] \sim, 3, 1, \ldots] 1]
? A = idealhnf(nf, 6*x, 4+2*x+x^2)
%3 =
[6 0 4]
[0 6 2]
[0 \ 0 \ 1]
? idealfactor(nf, A)
%4 =
 [[2, [0, 1, 0] \sim, 3, 1, \ldots] 2]
 [[3, [1, 1, 0] \sim, 3, 1, \ldots] 2]
? idealfactor(nf, A, 3) \setminus restrict to primes above p < 3
%5 =
[[2, [0, 1, 0]~, 3, 1, ...] 2]
```

idealfactorback(f, e, flag)

Gives back the ideal corresponding to a factorization. The integer 1 corresponds to the empty factorization. If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization, as produced by idealfactor.

```
? nf = nfinit(y^2+1); idealfactor(nf, 4 + 2*y)
%1 =
[[2, [1, 1]~, 2, 1, [1, 1]~] 2]
[[5, [2, 1]~, 1, 1, [-2, 1]~] 1]
? idealfactorback(nf, %)
%2 =
[10 4]
[0 2]
? f = %1[,1]; e = %1[,2]; idealfactorback(nf, f, e)
%3 =
[10 4]
[0 2]
```

? % == idealhnf(nf, 4 + 2*y) %4 = 1

If flag is nonzero, perform ideal reductions (idealred) along the way. This is most useful if the ideals involved are all *extended* ideals (for instance with trivial principal part), so that the principal parts extracted by idealred are not lost. Here is an example:

```
? f = vector(#f, i, [f[i], [;]]); \\ transform to extended ideals
? idealfactorback(nf, f, e, 1)
%6 = [[1, 0; 0, 1], [2, 1; [2, 1]~, 1]]
? nffactorback(nf, %[2])
%7 = [4, 2]~
```

The extended ideal returned in %6 is the trivial ideal 1, extended with a principal generator given in factored form. We use nffactorback to recover it in standard form.

idealfrobenius(gal, pr)

Let K be the number field defined by nf and assume K/\mathbb{Q} be a Galois extension with Galois group given gal = galoisinit(nf), and that pr is an unramified prime ideal p in prid format. This function returns a permutation of gal.group which defines the Frobenius element Frob_p attached to p. If p is the unique prime number in p, then $\operatorname{Frob}(x) = x^p \mod p$ for all $x \in \mathbb{Z}_K$.

? nf = nfinit(polcyclo(31)); ? gal = galoisinit(nf); ? pr = idealprimedec(nf,101)[1]; ? g = idealfrobenius(nf,gal,pr); ? galoispermtopol(gal,g) %5 = x^8

This is correct since 101 = 8mod31.

idealhnf(u, v)

Gives the Hermite normal form of the ideal $u\mathbb{Z}_K + v\mathbb{Z}_K$, where u and v are elements of the number field K defined by *nf*.

```
? nf = nfinit(y<sup>3</sup> - 2);
? idealhnf(nf, 2, y+1)
%2 =
[1 0 0]
[0 1 0]
[0 0 1]
? idealhnf(nf, y/2, [0,0,1/3]~)
%3 =
[1/3 0 0]
[0 1/6 0]
[0 0 1/6]
```

If *b* is omitted, returns the HNF of the ideal defined by *u*: *u* may be an algebraic number (defining a principal ideal), a maximal ideal (as given by idealprimedec or idealfactor), or a matrix whose columns give generators for

the ideal. This last format is a little complicated, but useful to reduce general modules to the canonical form once in a while:

- if strictly less than $N = [K : \mathbb{Q}]$ generators are given, u is the \mathbb{Z}_K -module they generate,
- if N or more are given, it is *assumed* that they form a Z-basis of the ideal, in particular that the matrix has maximal rank N. This acts as mathnf since the Z_K-module structure is (taken for granted hence) not taken into account in this case.

```
? idealhnf(nf, idealprimedec(nf,2)[1])
%4 =
[2 0 0]
[0 1 0]
[0 0 1]
? idealhnf(nf, [1,2;2,3;3,4])
%5 =
[1 0 0]
[0 1 0]
[0 0 1]
```

Finally, when K is quadratic with discriminant D_K , we allow u = Qfb(a,b,c), provided $b^2 - 4ac = D_K$. As usual, this represents the ideal $a\mathbb{Z} + (1/2)(-b + \sqrt{D_K})\mathbb{Z}$.

idealintersect(A, B)

Intersection of the two ideals A and B in the number field nf. The result is given in HNF.

```
? nf = nfinit(x^2+1);
? idealintersect(nf, 2, x+1)
%2 =
[2 0]
[0 2]
```

This function does not apply to general \mathbb{Z} -modules, e.g. orders, since its arguments are replaced by the ideals they generate. The following script intersects \mathbb{Z} -modules A and B given by matrices of compatible dimensions with integer coefficients:

```
ZM_intersect(A,B) =
{ my(Ker = matkerint(concat(A,B)));
```

mathnf(A * Ker[1..#A,])
}

idealinv(x)

Inverse of the ideal x in the number field nf, given in HNF. If x is an extended ideal, its principal part is suitably updated: i.e. inverting [I, t], yields $[I^{-1}, 1/t]$.

idealismaximal(x)

Given nf a number field as output by nfinit and an ideal x, return 0 if x is not a maximal ideal. Otherwise return a prid structure nf attached to the ideal. This function uses ispseudoprime and may return a wrong result in case the underlying rational pseudoprime is not an actual prime number: apply isprime(pr.p) to guarantee correctness. If x is an extended ideal, the extended part is ignored.

```
? K = nfinit(y<sup>2</sup> + 1);
? idealismaximal(K, 3) \\ 3 is inert
%2 = [3, [3, 0]~, 1, 2, 1]
? idealismaximal(K, 5) \\ 5 is not
%3 = 0
? pr = idealprimedec(K,5)[1] \\ already a prid
%4 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
? idealismaximal(K, pr) \\ trivial check
%5 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
? x = idealhnf(K, pr)
%6 =
[5 3]
[0 1]
? idealismaximal(K, x) \\ converts from matrix form to prid
%7 = [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]]
```

This function is noticeably faster than idealfactor since it never involves an actually factorization, in particular when $x \cap \mathbb{Z}$ is not a prime number.

idealispower(A, n, B)

Let *nf* be a number field and n > 0 be a positive integer. Return 1 if the fractional ideal $A = B^n$ is an *n*-th power and 0 otherwise. If the argument B is present, set it to the *n*-th root of A, in HNF.

```
? K = nfinit(x<sup>3</sup> - 2);
? A = [46875, 30966, 9573; 0, 3, 0; 0, 0, 3];
? idealispower(K, A, 3, &B)
%3 = 1
? B
%4 =
[75 22 41]
[ 0 1 0]
[ 0 0 1]
? A = [9375, 2841, 198; 0, 3, 0; 0, 0, 3];
? idealispower(K, A, 3)
%5 = 0
```

ideallist(bound, flag)

Computes the list of all ideals of norm less or equal to *bound* in the number field *nf*. The result is a row vector with exactly *bound* components. Each component is itself a row vector containing the information about ideals of a given norm, in no specific order, depending on the value of *flag*:

The possible values of flag are:

0: give the *bid* attached to the ideals, without generators.

1: as 0, but include the generators in the *bid*.

2: in this case, *nf* must be a *bnf* with units. Each component is of the form [bid, U], where *bid* is as case 0 and U is a vector of discrete logarithms of the units. More precisely, it gives the ideallog s with respect to *bid* of $(\zeta, u_1, ..., u_r)$ where ζ is the torsion unit generator bnf.tu[2] and (u_i) are the fundamental units in bnf.fu. This structure is technical, and only meant to be used in conjunction with bnrclassnolist or bnrdisclist.

3: as 2, but include the generators in the *bid*.

4: give only the HNF of the ideal.

```
? nf = nfinit(x^2+1);
? L = ideallist(nf, 100);
? L[1]
%3 = [[1, 0; 0, 1]] \\ A single ideal of norm 1
? #L[65]
%4 = 4 \\ There are 4 ideals of norm 4 in Z[i]
```

If one wants more information, one could do instead:

```
? nf = nfinit(x^2+1);
? L = ideallist(nf, 100, 0);
? l = L[25]; vector(#l, i, 1[i].clgp)
%3 = [[20, [20]], [16, [4, 4]], [20, [20]]]
? 1[1].mod
%4 = [[25, 18; 0, 1], []]
? 1[2].mod
%5 = [[5, 0; 0, 5], []]
? 1[3].mod
%6 = [[25, 7; 0, 1], []]
```

where we ask for the structures of the $(\mathbb{Z}[i]/I)^*$ for all three ideals of norm 25. In fact, for all moduli with finite part of norm 25 and trivial Archimedean part, as the last 3 commands show. See ideallistarch to treat general moduli.

ideallistarch(list, arch)

list is a vector of vectors of bid's, as output by ideallist with flag 0 to 3. Return a vector of vectors with the same number of components as the original *list*. The leaves give information about moduli whose finite part is as in original list, in the same order, and Archimedean part is now *arch* (it was originally trivial). The information contained is of the same kind as was present in the input; see ideallist, in particular the meaning of *flag*.

```
? bnf = bnfinit(x^2-2);
? bnf.sign
%2 = [2, 0] \\ two places at infinity
? L = ideallist(bnf, 100, 0);
? l = L[98]; vector(#l, i, l[i].clgp)
%4 = [[42, [42]], [36, [6, 6]], [42, [42]]]
? La = ideallistarch(bnf, L, [1,1]); \\ add them to the modulus
```

```
? l = La[98]; vector(#l, i, l[i].clgp)
%6 = [[168, [42, 2, 2]], [144, [6, 6, 2, 2]], [168, [42, 2, 2]]]
```

Of course, the results above are obvious: adding t places at infinity will add t copies of $\mathbb{Z}/2\mathbb{Z}$ to $(\mathbb{Z}_K/f)^*$. The following application is more typical:

```
? L = ideallist(bnf, 100, 2); \\ units are required now
? La = ideallistarch(bnf, L, [1,1]);
? H = bnrclassnolist(bnf, La);
? H[98];
%4 = [2, 12, 2]
```

ideallog(x, bid)

nf is a number field, *bid* is as output by idealstar(nf, D,...) and x an element of nf which must have valuation equal to 0 at all prime ideals in the support of D and need not be integral. This function computes the discrete logarithm of x on the generators given in :emphasis:`bid.gen`. In other words, if g_i are these generators, of orders d_i respectively, the result is a column vector of integers (x_i) such that $0 \le x_i < d_i$ and

$$x = \prod_{i} g_i^{x_i}(mod^*D).$$

Note that when the support of D contains places at infinity, this congruence implies also sign conditions on the attached real embeddings. See znlog for the limitations of the underlying discrete log algorithms.

When nf is omitted, take it to be the rational number field. In that case, x must be a t_INT and *bid* must have been initialized by znstar(N, 1).

idealmin(ix, vdir)

This function is useless and kept for backward compatibility only, use :literal: `idealred`. Computes a pseudominimum of the ideal x in the direction vdir in the number field nf.

idealmul(x, y, flag)

Ideal multiplication of the ideals x and y in the number field nf; the result is the ideal product in HNF. If either x or y are extended ideals, their principal part is suitably updated: i.e. multiplying [I, t], [J, u] yields [IJ, tu]; multiplying I and [J, u] yields [IJ, u].

```
? nf = nfinit(x^2 + 1);
? idealmul(nf, 2, x+1)
%2 =
[4 2]
[0 2]
? idealmul(nf, [2, x], x+1) \\ extended ideal * ideal
%3 = [[4, 2; 0, 2], x]
? idealmul(nf, [2, x], [x+1, x]) \\ two extended ideals
%4 = [[4, 2; 0, 2], [-1, 0]~]
```

If *flag* is nonzero, reduce the result using idealred.

idealnorm(x)

Computes the norm of the ideal x in the number field nf.

idealnumden(x)

Returns [A, B], where A, B are coprime integer ideals such that x = A/B, in the number field nf.

? nf = nfinit(x^2+1); ? idealnumden(nf, (x+1)/2) %2 = [[1, 0; 0, 1], [2, 1; 0, 1]]

idealpow(x, k, flag)

Computes the k-th power of the ideal x in the number field nf; $k \in \mathbb{Z}$. If x is an extended ideal, its principal part is suitably updated: i.e. raising [I, t] to the k-th power, yields $[I^k, t^k]$.

If flag is nonzero, reduce the result using idealred, throughout the (binary) powering process; in particular, this is not the same as idealpow(nf, x, k) followed by reduction.

idealprimedec(p,f)

Computes the prime ideal decomposition of the (positive) prime number p in the number field K represented by *nf*. If a nonprime p is given the result is undefined. If f is present and nonzero, restrict the result to primes of residue degree $\leq f$.

The result is a vector of *prid* structures, each representing one of the prime ideals above p in the number field nf. The representation pr = [p, a, e, f, mb] of a prime ideal means the following: a is an algebraic integer in the maximal order \mathbb{Z}_K and the prime ideal is equal to $p = p\mathbb{Z}_K + a\mathbb{Z}_K$; e is the ramification index; f is the residual index; finally, mb is the multiplication table attached to the algebraic integer b is such that $p^{-1} = \mathbb{Z}_K + b/p\mathbb{Z}_K$, which is used internally to compute valuations. In other words if p is inert, then mb is the integer 1, and otherwise it is a square t_MAT whose j-th column is b.nf.zk[j].

The algebraic number a is guaranteed to have a valuation equal to 1 at the prime ideal (this is automatic if e > 1).

The components of pr should be accessed by member functions: pr.p, pr.e, pr.f, and pr.gen (returns the vector [p, a]):

```
? K = nfinit(x^3-2);
? P = idealprimedec(K, 5);
? #P \\ 2 primes above 5 in Q(2^(1/3))
%3 = 2
? [p1,p2] = P;
? [p1.e, p1.f] \\ the first is unramified of degree 1
%5 = [1, 1]
? [p2.e, p2.f] \\ the second is unramified of degree 2
%6 = [1, 2]
? p1.gen
%7 = [5, [2, 1, 0]~]
? nfbasistoalg(K, %[2]) \\ a uniformizer for p1
%8 = Mod(x + 2, x^3 - 2)
? #idealprimedec(K, 5, 1) \\ restrict to f = 1
%9 = 1 \\ now only p1
```

idealprincipalunits(pr, k)

Given a prime ideal in idealprimedec format, returns the multiplicative group $(1+pr)/(1+pr^k)$ as an abelian group. This function is much faster than idealstar when the norm of pr is large, since it avoids (useless) work in the multiplicative group of the residue field.

```
? K = nfinit(y^2+1);
? P = idealprimedec(K,2)[1];
? G = idealprincipalunits(K, P, 20);
? G.cyc
%4 = [512, 256, 4] \\ Z/512 x Z/256 x Z/4
```

? G.gen %5 = [[-1, -2]~, 1021, [0, -1]~] \\ minimal generators of given order

idealramgroups(gal, pr)

Let K be the number field defined by nf and assume that K/\mathbb{Q} is Galois with Galois group G given by gal = galoisinit(nf). Let pr be the prime ideal P in prid format. This function returns a vector g of subgroups of gal as follows:

- g[1] is the decomposition group of P,
- g[2] is $G_0(P)$, the inertia group of P,

and for $i \ge 2$,

• g[i] is $G_{i-2}(P)$, the i-2-th ramification group of P.

The length of g is the number of nontrivial groups in the sequence, thus is 0 if e = 1 and f = 1, and 1 if f > 1 and e = 1. The following function computes the cardinality of a subgroup of G, as given by the components of g:

card(H) =my(o=H[2]); prod(i=1,#o,o[i]);

```
? nf=nfinit(x^6+3); gal=galoisinit(nf); pr=idealprimedec(nf,3)[1];
? g = idealramgroups(nf, gal, pr);
? apply(card,g)
%3 = [6, 6, 3, 3, 3] \\ cardinalities of the G_i
```

```
? nf=nfinit(x^6+108); gal=galoisinit(nf); pr=idealprimedec(nf,2)[1];
? iso=idealramgroups(nf,gal,pr)[2]
%5 = [[Vecsmall([2, 3, 1, 5, 6, 4])], Vecsmall([3])]
? nfdisc(galoisfixedfield(gal,iso,1))
%6 = -3
```

The field fixed by the inertia group of 2 is not ramified at 2.

idealred(I, v)

LLL reduction of the ideal I in the number field K attached to nf, along the direction v. The v parameter is best left omitted, but if it is present, it must be an nf.r1 + nf.r2-component vector of *nonnegative* integers. (What counts is the relative magnitude of the entries: if all entries are equal, the effect is the same as if the vector had been omitted.)

This function finds an $a \in K^*$ such that J = (a)I is "small" and integral (see the end for technical details). The result is the Hermite normal form of the "reduced" ideal J.

? K = nfinit(y²+1); ? P = idealprimedec(K,5)[1]; ? idealred(K, P) %3 = [1 0] [0 1]

More often than not, a principal ideal yields the unit ideal as above. This is a quick and dirty way to check if ideals are principal, but it is not a necessary condition: a nontrivial result does not prove that the ideal is nonprincipal. For guaranteed results, see **bnfisprincipal**, which requires the computation of a full **bnf** structure.

If the input is an extended ideal [I, s], the output is [J, sa]; in this way, one keeps track of the principal ideal part:

meaning that P is generated by [2, -1]. The number field element in the extended part is an algebraic number in any form *or* a factorization matrix (in terms of number field elements, not ideals!). In the latter case, elements stay in factored form, which is a convenient way to avoid coefficient explosion; see also idealpow.

Technical note. The routine computes an LLL-reduced basis for the lattice I^{-1} equipped with the quadratic form

$$|||x|||_{v}^{2} = \sum_{i=1}^{r_{1}+r_{2}} 2^{v_{i}} \varepsilon_{i} ||\sigma_{i}(x)||^{2},$$

where as usual the σ_i are the (real and) complex embeddings and $\varepsilon_i = 1$, resp. 2, for a real, resp. complex place. The element *a* is simply the first vector in the LLL basis. The only reason you may want to try to change some directions and set some $v_i! = 0$ is to randomize the elements found for a fixed ideal, which is heuristically useful in index calculus algorithms like bnfinit and bnfisprincipal.

Even more technical note. In fact, the above is a white lie. We do not use $||||.|||_v$ exactly but a rescaled rounded variant which gets us faster and simpler LLLs. There's no harm since we are not using any theoretical property of *a* after all, except that it belongs to I^{-1} and that aI is "expected to be small".

idealredmodpower(x, n, B)

Let *nf* be a number field, x an ideal in *nf* and n > 0 be a positive integer. Return a number field element b such that $xb^n = v$ is small. If x is integral, then v is also integral.

More precisely, idealnumden reduces the problem to x integral. Then, factoring out the prime ideals dividing a rational prime $p \le B$, we rewrite $x = IJ^n$ where the ideals I and J are both integral and I is B-smooth. Then we return a small element b in J^{-1} .

The bound B avoids a costly complete factorization of x; as soon as the *n*-core of x is B-smooth (i.e., as soon as I is *n*-power free), then J is as large as possible and so is the expected reduction.

```
? T = x^6+108; nf = nfinit(T); a = Mod(x,T);
? setrand(1); u = (2*a^2+a+3)*random(2^1000*x^6)^6;
? sizebyte(u)
%3 = 4864
? b = idealredmodpower(nf,u,2);
? v2 = nfeltmul(nf,u, nfeltpow(nf,b,2))
%5 = [34, 47, 15, 35, 9, 3]~
? b = idealredmodpower(nf,u,6);
? v6 = nfeltmul(nf,u, nfeltpow(nf,b,6))
%7 = [3, 0, 2, 6, -7, 1]~
```

The last element v6, obtained by reducing modulo 6-th powers instead of squares, looks smaller than v2 but its norm is actually a little larger:

```
? idealnorm(nf,v2)
%8 = 81309
? idealnorm(nf,v6)
%9 = 731781
```

idealstar(N, flag, cycmod)

Outputs a bid structure, necessary for computing in the finite abelian group $G = (\mathbb{Z}_K/N)^*$. Here, *nf* is a number field and N is a *modulus*: either an ideal in any form, or a row vector whose first component is an ideal and whose

second component is a row vector of r_1 0 or 1. Ideals can also be given by a factorization into prime ideals, as produced by idealfactor.

If the positive integer cycmod is present, only compute the group modulo cycmod-th powers, which may save a lot of time when some maximal ideals in the modulus have a huge residue field. Whereas you might only be interested in quadratic or cubic residuosity; see also **bnrinit** for applications in class field theory.

This *bid* is used in ideallog to compute discrete logarithms. It also contains useful information which can be conveniently retrieved as :emphasis:`bid.mod` (the modulus), :emphasis:`bid.clgp` (G as a finite abelian group), :emphasis:`bid.no` (the cardinality of G), :emphasis:`bid.cyc` (elementary divisors) and :emphasis:`bid.gen` (generators).

If flag = 1 (default), the result is a bid structure without generators: they are well defined but not explicitly computed, which saves time.

If flag = 2, as flag = 1, but including generators.

If flag = 0, only outputs $(\mathbb{Z}_K/N)^*$ as an abelian group, i.e as a 3-component vector [h, d, g]: h is the order, d is the vector of SNF cyclic components and g the corresponding generators.

If *nf* is omitted, we take it to be the rational number fields, N must be an integer and we return the structure of $(\mathbb{Z}/N\mathbb{Z})^*$. In other words idealstar(, N, flag) is short for

idealstar(nfinit(x), N, flag)

but faster. The alternative syntax znstar(N, flag) is also available for an analogous effect but, due to an unfortunate historical oversight, the default value of flag is different in the two functions (znstar does not initialize by default, you probably want znstar(N, 1)).

idealtwoelt(x, a)

Computes a two-element representation of the ideal x in the number field nf, combining a random search and an approximation theorem; x is an ideal in any form (possibly an extended ideal, whose principal part is ignored)

• When called as idealtwoelt(nf,x), the result is a row vector $[a, \alpha]$ with two components such that $x = a\mathbb{Z}_K + \alpha\mathbb{Z}_K$ and a is chosen to be the positive generator of $x \cap \mathbb{Z}$, unless x was given as a principal ideal in which case we may choose a = 0. The algorithm uses a fast lazy factorization of $x \cap \mathbb{Z}$ and runs in randomized polynomial time.

```
? K = nfinit(t^5-23);
? x = idealhnf(K, t^2*(t+1), t^3*(t+1))
%2 = \\ some random ideal of norm 552*23
[552 23 23 529 23]
[ 0 23 0 0 0]
[ 0 0 1 0 0]
[ 0 0 0 1 0]
[ 0 0 0 1 0]
[ 0 0 0 0 1]
? [a,alpha] = idealtwoelt(K, x)
%3 = [552, [23, 0, 1, 0, 0]~]
? nfbasistoalg(K, alpha)
%4 = Mod(t^2 + 23, t^5 - 23)
```

• When called as idealtwoelt(nf,x,a) with an explicit nonzero *a* supplied as third argument, the function assumes that $a \in x$ and returns $\alpha \in x$ such that $x = a\mathbb{Z}_K + \alpha\mathbb{Z}_K$. Note that we must factor *a* in this case,

and the algorithm is generally slower than the default variant and gives larger generators:

```
? alpha2 = idealtwoelt(K, x, 552)
%5 = [-161, -161, -183, -207, 0]~
? idealhnf(K, 552, alpha2) == x
%6 = 1
```

Note that, in both cases, the return value is *not* recognized as an ideal by GP functions; one must use idealhnf as above to recover a valid ideal structure from the two-element representation.

idealval(x, pr)

Gives the valuation of the ideal x at the prime ideal pr in the number field nf, where pr is in idealprimedec format. The valuation of the 0 ideal is +00.

imag()

Imaginary part of x. When x is a quadratic number, this is the coefficient of ω in the "canonical" integral basis $(1, \omega)$.

? imag(3 + I) %1 = 1 ? x = 3 + quadgen(-23); ? imag(x) \\ as a quadratic number %3 = 1 ? imag(x * 1.) \\ as a complex number %4 = 2.3979157616563597707987190320813469600

incgam(*x*, *g*, *precision*)

Incomplete gamma function $\int_x^o oe^{-t}t^{s-1}dt$, extended by analytic continuation to all complex x, s not both 0. The relative error is bounded in terms of the precision of s (the accuracy of x is ignored when determining the output precision). When g is given, assume that $g = \Gamma(s)$. For small ||x||, this will speed up the computation.

incgamc(*x*, *precision*)

Complementary incomplete gamma function. The arguments x and s are complex numbers such that s is not a pole of Γ and ||x||/(||s|| + 1) is not much larger than 1 (otherwise the convergence is very slow). The result returned is $\int_0^x e^{-t} t^{s-1} dt$.

intformal(v)

formal integration of x with respect to the variable v (wrt. the main variable if v is omitted). Since PARI cannot represent logarithmic or arctangent terms, any such term in the result will yield an error:

```
? intformal(x^2)
%1 = 1/3*x^3
? intformal(x^2, y)
%2 = y*x^2
? intformal(1/x)
*** at top-level: intformal(1/x)
*** ^------
*** intformal: domain error in intformal: residue(series, pole) != 0
```

The argument x can be of any type. When x is a rational function, we assume that the base ring is an integral domain of characteristic zero.

By definition, the main variable of a t_POLMOD is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of R[X]/(T(X)), the variable X is a mute variable and the integral is taken with respect to the main variable used in the base ring R. In particular, it is meaningless to integrate with respect to the main variable of x.mod:
```
? intformal(Mod(1,x^2+1), 'x)
*** intformal: incorrect priority in intformal: variable x = x
```

intnuminit(b, m, precision)

Initialize tables for integration from *a* to *b*, where *a* and *b* are coded as in intrum. Only the compactness, the possible existence of singularities, the speed of decrease or the oscillations at infinity are taken into account, and not the values. For instance intruminit((-1, 1) is equivalent to intruminit((0, -1/2], 00) is equivalent to intruminit((-1, -1/2], -00); on the other hand, the order matters and intruminit((0, -1/2], (1, -1/3]) is *not* equivalent to intruminit((0, -1/3], (1, -1/2])!

If m is present, it must be nonnegative and we multiply the default number of sampling points by 2^m (increasing the running time by a similar factor).

The result is technical and liable to change in the future, but we document it here for completeness. Let $x = \phi(t)$, $t \in]-oo, oo[$ be an internally chosen change of variable, achieving double exponential decrease of the integrand at infinity. The integrator intrum will compute

$$h\sum_{\|n\| < N} \phi'(nh) F(\phi(nh))$$

for some integration step h and truncation parameter N. In basic use, let

[h, x0, w0, xp, wp, xm, wm] = intnuminit(a,b);

- *h* is the integration step
- $x_0 = \phi(0)$ and $w_0 = \phi'(0)$,
- *xp* contains the $\phi(nh)$, 0 < n < N,
- xm contains the $\phi(nh)$, 0 < -n < N, or is empty.
- wp contains the $\phi'(nh)$, 0 < n < N,
- wm contains the $\phi'(nh)$, 0 < -n < N, or is empty.

The arrays xm and wm are left empty when ϕ is an odd function. In complicated situations, intruminit may return up to 3 such arrays, corresponding to a splitting of up to 3 integrals of basic type.

If the functions to be integrated later are of the form F = f(t)k(t, z) for some kernel k (e.g. Fourier, Laplace, Mellin,...), it is useful to also precompute the values of $f(\phi(nh))$, which is accomplished by intfuncinit. The hard part is to determine the behavior of F at endpoints, depending on z.

isfundamental()

True (1) if D is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise. D can be input in factored form as for arithmetic functions:

```
? isfundamental(factor(-8))
%1 = 1
\\ count fundamental discriminants up to 10^8
? c = 0; forfactored(d = 1, 10^8, if (isfundamental(d), c++)); c
time = 40,840 ms.
%2 = 30396325
? c = 0; for(d = 1, 10^8, if (isfundamental(d), c++)); c
time = 1min, 33,593 ms. \\ slower !
%3 = 30396325
```

ispolygonal(s, N)

True (1) if the integer x is an s-gonal number, false (0) if not. The parameter s > 2 must be a t_INT. If N is given, set it to n if x is the n-th s-gonal number.

? ispolygonal(36, 3, &N) %1 = 1 ? N

ispower(k, n)

If k is given, returns true (1) if x is a k-th power, false (0) if not. What it means to be a k-th power depends on the type of x; see issquare for details.

If k is omitted, only integers and fractions are allowed for x and the function returns the maximal $k \ge 2$ such that $x = n^k$ is a perfect power, or 0 if no such k exist; in particular ispower(-1), ispower(0), and ispower(1) all return 0.

If a third argument n is given and x is indeed a k-th power, sets n to a k-th root of x.

For a t_FFELT x, instead of omitting k (which is not allowed for this type), it may be natural to set

 $\mathbf{k} = (\mathbf{x} \cdot \mathbf{p} \wedge \mathbf{x} \cdot \mathbf{f} - 1) / \text{fforder}(\mathbf{x})$

ispowerful()

True (1) if x is a powerful integer, false (0) if not; an integer is powerful if and only if its valuation at all primes dividing x is greater than 1.

```
? ispowerful(50)
%1 = 0
? ispowerful(100)
%2 = 1
? ispowerful(5^3*(10^1000+1)^2)
%3 = 1
```

isprime(flag)

True (1) if x is a prime number, false (0) otherwise. A prime number is a positive integer having exactly two distinct divisors among the natural numbers, namely 1 and itself.

This routine proves or disproves rigorously that a number is prime, which can be very slow when x is indeed a large prime integer. For instance a 1000 digits prime should require 15 to 30 minutes with default algorithms. Use ispseudoprime to quickly check for compositeness. Use primecert in order to obtain a primality proof instead of a yes/no answer; see also factor.

The function accepts vector/matrices arguments, and is then applied componentwise.

If flag = 0, use a combination of

- Baillie-Pomerance-Selfridge-Wagstaff compositeness test (see ispseudoprime),
- Selfridge "p 1" test if x 1 is smooth enough,
- Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) for general medium-sized x (less than 1500 bits),
- Atkin-Morain's Elliptic Curve Primality Prover (ECPP) for general large x.

If flag = 1, use Selfridge-Pocklington-Lehmer "p - 1" test; this requires partially factoring various auxilliary integers and is likely to be very slow.

If flag = 2, use APRCL only.

If flag = 3, use ECPP only.

isprimepower(n)

If $x = p^k$ is a prime power (p prime, k > 0), return k, else return 0. If a second argument n is given and x is indeed the k-th power of a prime p, sets n to p.

ispseudoprime(flag)

True (1) if x is a strong pseudo prime (see below), false (0) otherwise. If this function returns false, x is not prime; if, on the other hand it returns true, it is only highly likely that x is a prime number. Use *isprime* (which is of course much slower) to prove that x is indeed prime. The function accepts vector/matrices arguments, and is then applied componentwise.

If flag = 0, checks whether x has no small prime divisors (up to 101 included) and is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime. Such a pseudo prime passes a Rabin-Miller test for base 2, followed by a Lucas test for the sequence (P, 1), where $P \ge 3$ is the smallest odd integer such that $P^2 - 4$ is not a square mod x. (Technically, we are using an "almost extra strong Lucas test" that checks whether V_n is 2, without computing U_{n} .)

There are no known composite numbers passing the above test, although it is expected that infinitely many such numbers exist. In particular, all composites $\leq 2^{64}$ are correctly detected (checked using http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html).

If flag > 0, checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

ispseudoprimepower(n)

If $x = p^k$ is a pseudo-prime power (p pseudo-prime as per ispseudoprime, k > 0), return k, else return 0. If a second argument n is given and x is indeed the k-th power of a prime p, sets n to p.

More precisely, k is always the largest integer such that $x = n^k$ for some integer n and, when $n <= 2^{64}$ the function returns k > 0 if and only if n is indeed prime. When $n > 2^{64}$ is larger than the threshold, the function may return 1 even though n is composite: it only passed an ispseudoprime(n) test.

issquare(n)

True (1) if x is a square, false (0) if not. What "being a square" means depends on the type of x: all t_COMPLEX are squares, as well as all nonnegative t_REAL; for exact types such as t_INT, t_FRAC and t_INTMOD, squares are numbers of the form s^2 with s in \mathbb{Z} , \mathbb{Q} and $\mathbb{Z}/N\mathbb{Z}$ respectively.

```
? issquare(3) \\ as an integer
%1 = 0
? issquare(3.) \\ as a real number
%2 = 1
? issquare(Mod(7, 8)) \\ in Z/8Z
%3 = 0
? issquare( 5 + 0(13<sup>4</sup>) ) \\ in Q_13
%4 = 0
```

If n is given, a square root of x is put into n.

```
? issquare(4, &n)
%1 = 1
? n
%2 = 2
```

For polynomials, either we detect that the characteristic is 2 (and check directly odd and even-power monomials) or we assume that 2 is invertible and check whether squaring the truncated power series for the square root yields the original input.

For t_POLMOD x, we only support t_POLMOD s of t_INTMOD s encoding finite fields, assuming without checking that the intmod modulus p is prime and that the polmod modulus is irreducible modulo p.

? issquare(Mod(Mod(2,3), x^2+1), &n)
%1 = 1
? n
%2 = Mod(Mod(2, 3)*x, Mod(1, 3)*x^2 + Mod(1, 3))

issquarefree()

True (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial with coefficients in an integral domain.

```
? issquarefree(12)
%1 = 0
? issquarefree(6)
%2 = 1
? issquarefree(x^3+x^2)
%3 = 0
? issquarefree(Mod(1,4)*(x^2+x+1)) \\ Z/4Z is not a domain !
*** at top-level: issquarefree(Mod(1,4)*(x^2+x+1))
*** ^------
*** issquarefree: impossible inverse in Fp_inv: Mod(2, 4).
```

A polynomial is declared squarefree if gcd(x, x') is 1. In particular a nonzero polynomial with inexact coefficients is considered to be squarefree. Note that this may be inconsistent with factor, which first rounds the input to some exact approximation before factoring in the apropriate domain; this is correct when the input is not close to an inseparable polynomial (the resultant of x and x' is not close to 0).

An integer can be input in factored form as in arithmetic functions.

```
? issquarefree(factor(6))
%1 = 1
\\ count squarefree integers up to 10^8
? c = 0; for(d = 1, 10^8, if (issquarefree(d), c++)); c
time = 3min, 2,590 ms.
%2 = 60792694
? c = 0; forfactored(d = 1, 10^8, if (issquarefree(d), c++)); c
time = 45,348 ms. \\ faster !
%3 = 60792694
```

istotient(N)

True (1) if $x = \phi(n)$ for some integer n, false (0) if not.

```
? istotient(14)
%1 = 0
? istotient(100)
%2 = 0
```

If N is given, set N = n as well.

? istotient(4, &n) %1 = 1 ? n %2 = 10

kronecker(y)

Kronecker symbol (x||y), where x and y must be of type integer. By definition, this is the extension of Legendre

symbol to $\mathbb{Z}x\mathbb{Z}$ by total multiplicativity in both arguments with the following special rules for y = 0, -1 or 2:

- (x||0) = 1 if ||x|| = 1 and 0 otherwise.
- $(x \parallel -1) = 1$ if $x \ge 0$ and -1 otherwise.
- (x||2) = 0 if x is even and 1 if $x = 1, -1 \mod 8$ and -1 if $x = 3, -3 \mod 8$.

lambertw(precision)

Lambert W function, solution of the implicit equation $xe^x = y$, for a positive real number y. This is the restriction to the positive reals of the complex principal branch W_0 , which is not implemented outside of \mathbb{R}^*_+ . Other branches W_k for k! = 0 are not implemented either.

laurentseries(serprec, M, precision)

Expand f as a Laurent series around x = 0 to order M. This function computes $f(x + O(x^n))$ until n is large enough: it must be possible to evaluate f on a power series with 0 constant term.

Note that individual Laurent coefficients of order $\leq M$ can be retrieved from s = laurentseries(f, M) via polcoef(s,i) for any $i \leq M$. The series s may occasionally be more precise that the required $O(x^{M+1})$.

With respect to successive calls to derivnum, laurentseries is both faster and more precise:

```
? laurentseries(t->log(3+t),1)
\%1 = 1.0986122886681096913952452369225257047 + 1/3*x - 1/18*x^2 + 0(x^3)
? derivnum(t=0, \log(3+t), 1)
? derivnum(t=0, \log(3+t), 2)
? f = x - sin(exp(x));
? polcoef(laurentseries(x->f(x+2), 1), 1)
%5 = 3.3129294231043339804683687620360224365
? exp(2) * cos(exp(2));
%6 = 3.3129294231043339804683687620360224365
? derivnum(x = 2, f(x))
%7 = 3.3129294231043339804683687620360224364 \\ 1 ulp off
? default(realprecision,115);
? for(i=1,10^4, laurentseries(x->f(x+2),1))
time = 279 \text{ ms.}
? for(i=1,10^4, derivnum(x=2,f(x))) \setminus ... and slower
time = 1.134 ms.
```

lcm(y)

Least common multiple of x and y, i.e. such that lcm(x, y) * gcd(x, y) = x * y, up to units. If y is omitted and x is a vector, returns the *lcm* of all components of x. For integer arguments, return the nonnegative lcm.

When x and y are both given and one of them is a vector/matrix type, the LCM is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to lcm(x, y[i]), resp. lcm(x, y[,i]). Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, lcm is not commutative.

Note that lcm(v) is quite different from

l = v[1]; for (i = 1, #v, l = lcm(l, v[i]))

Indeed, lcm(v) is a scalar, but 1 may not be (if one of the v[i] is a vector/matrix). The computation uses a divide-conquer tree and should be much more efficient, especially when using the GMP multiprecision kernel (and more subquadratic algorithms become available):

? v = vector(10^5, i, random); ? lcm(v); time = 546 ms. ? l = v[1]; for (i = 1, #v, l = lcm(l, v[i])) time = 4,561 ms.

length()

Length of x; #x is a shortcut for length(x). This is mostly useful for

- vectors: dimension (0 for empty vectors),
- lists: number of entries (0 for empty lists),
- maps: number of entries (0 for empty maps),
- matrices: number of columns,
- character strings: number of actual characters (without trailing 0, should you expect it from C char*).

? #"a string"
%1 = 8
? #[3,2,1]
%2 = 3
? #[]
%3 = 0
? #matrix(2,5)
%4 = 5
? L = List([1,2,3,4]); #L
%5 = 4
? M = Map([a,b; c,d; e,f]); #M
%6 = 3

The routine is in fact defined for arbitrary GP types, but is awkward and useless in other cases: it returns the number of non-code words in x, e.g. the effective length minus 2 for integers since the t_INT type has two code words.

lex(y)

Gives the result of a lexicographic comparison between x and y (as -1, 0 or 1). This is to be interpreted in quite a wide sense: it is admissible to compare objects of different types (scalars, vectors, matrices), provided the scalars can be compared, as well as vectors/matrices of different lengths; finally, when comparing two scalars, a complex number a + I * b is interpreted as a vector [a, b] and a real number a as [a, 0]. The comparison is recursive.

In case all components are equal up to the smallest length of the operands, the more complex is considered to be larger. More precisely, the longest is the largest; when lengths are equal, we have matrix > vector > scalar. For example:

? lex([1,3], [1,2,5]) %1 = 1 ? lex([1,3], [1,3,-1]) %2 = -1 ? lex([1], [[1]]) %3 = -1 ? lex([1], [1]~) %4 = 0 ? lex(2 - I, 1) %5 = 1 ? lex(2 - I, 2) %6 = 2

lfun(*s*, *D*, *precision*)

Compute the L-function value L(s), or if D is set, the derivative of order D at s. The parameter L is either an Lmath, an Ldata (created by lfuncreate, or an Linit (created by lfuninit), preferrably the latter if many values are to be computed.

The argument s is also allowed to be a power series; for instance, if $s = \alpha + x + O(x^n)$, the function returns the Taylor expansion of order n around α . The result is given with absolute error less than 2^{-B} , where B = realbit precision.

Caveat. The requested precision has a major impact on runtimes. It is advised to manipulate precision via realbitprecision as explained above instead of realprecision as the latter allows less granularity: realprecision increases by increments of 64 bits, i.e. 19 decimal digits at a time.

```
? lfun(x^2+1, 2) \\ Lmath: Dedekind zeta for Q(i) at 2
%1 = 1.5067030099229850308865650481820713960
? L = lfuncreate(ellinit("5077a1")); \\ Ldata: Hasse-Weil zeta function
? lfun(L, 1+x+0(x^4)) \\ zero of order 3 at the central point
%3 = 0.E-58 - 5.[...] E-40*x + 9.[...] E-40*x^2 + 1.7318[...]*x^3 + 0(x^4)
\\ Linit: zeta(1/2+it), |t| < 100, and derivative
? L = lfuninit(1, [100], 1);
? T = lfunzeros(L, [1,25]);
%5 = [14.134725[...], 21.022039[...]]
? z = 1/2 + I*T[1];
? abs( lfun(L, z) )
%7 = 8.7066865533412207420780392991125136196 E-39
? abs( lfun(L, z, 1) )
%8 = 0.79316043335650611601389756527435211412 \\ simple zero
```

lfunabelianrelinit(bnfK, polrel, sdom, der, precision)

Returns the Linit structure attached to the Dedekind zeta function of the number field L (see lfuninit), given a subfield K such that L/K is abelian. Here polrel defines L over K, as usual with the priority of the variable of bnfK lower than that of polrel. sdom and der are as in lfuninit.

```
? D = -47; K = bnfinit(y^2-D);
? rel = quadhilbert(D); T = rnfequation(K.pol, rel); \\ degree 10
? L = lfunabelianrelinit(T,K,rel, [2,0,0]); \\ at 2
time = 84 ms.
? lfun(L, 2)
%4 = 1.0154213394402443929880666894468182650
```

```
? lfun(T, 2) \\ using parisize > 300MB
time = 652 ms.
%5 = 1.0154213394402443929880666894468182656
```

As the example shows, using the (abelian) relative structure is more efficient than a direct computation. The difference becomes drastic as the absolute degree increases while the subfield degree remains constant.

lfunan(*n*, *precision*)

Compute the first *n* terms of the Dirichlet series attached to the *L*-function given by L (Lmath, Ldata or Linit).

```
? lfunan(1, 10) \\ Riemann zeta
%1 = [1, 1, 1, 1, 1, 1, 1, 1, 1]
? lfunan(5, 10) \\ Dirichlet L-function for kronecker(5,.)
%2 = [1, -1, -1, 1, 0, 1, -1, -1, 1, 0]
```

lfunartin(gal, rho, n, precision)

Returns the Ldata structure attached to the Artin L-function provided by the representation ρ of the Galois group of the extension K/\mathbb{Q} , defined over the cyclotomic field $\mathbb{Q}(\zeta_n)$, where *nf* is the nfinit structure attached to *K*, *gal* is the galoisinit structure attached to K/\mathbb{Q} , and *rho* is given either

- by the values of its character on the conjugacy classes (see galoisconjclasses and galoischartable)
- or by the matrices that are the images of the generators :emphasis:`gal.gen`.

Cyclotomic numbers in rho are represented by polynomials, whose variable is understood as the complex number $\exp(2i\pi/n)$.

In the following example we build the Artin L-functions attached to the two irreducible degree 2 representations of the dihedral group D_{10} defined over $\mathbb{Q}(\zeta_5)$, for the extension H/\mathbb{Q} where H is the Hilbert class field of $\mathbb{Q}(\sqrt{-47})$. We show numerically some identities involving Dedekind ζ functions and Hecke L series.

```
? P = quadhilbert(-47)
\%1 = x^5 + 2x^4 + 2x^3 + x^2 - 1
? N = nfinit(nfsplitting(P));
? G = galoisinit(N); \setminus D_{10}
? [T,n] = galoischartable(G);
? T \\ columns give the irreducible characters
%5 =
[1 \ 1 \ 2 \ 2]
[1 - 1 \ 0 \ 0]
[1 \ 1 \ -y^3 \ -y^2 \ -1 \ y^3 \ +y^2]
[1 \ 1 \ y^3 + y^2 - y^3 - y^2 - 1]
? n
%6 = 5
? L2 = lfunartin(N,G, T[,2], n);
? L3 = lfunartin(N,G, T[,3], n);
? L4 = lfunartin(N,G, T[,4], n);
? s = 1 + x + 0(x^4);
? lfun(-47,s) - lfun(L2,s)
%11 ~ 0
? lfun(1,s)*lfun(-47,s)*lfun(L3,s)^2*lfun(L4,s)^2 - lfun(N,s)
```

```
%12 ~ 0
? lfun(1,s)*lfun(L3,s)*lfun(L4,s) - lfun(P,s)
%13 ~ 0
? bnr = bnrinit(bnfinit(x^2+47),1,1);
? bnr.cyc
%15 = [5] \\ Z/5Z: 4 nontrivial ray class characters
? lfun([bnr,[1]], s) - lfun(L3, s)
%16 ~ 0
? lfun([bnr,[2]], s) - lfun(L4, s)
%17 ~ 0
? lfun([bnr,[3]], s) - lfun(L3, s)
%18 ~ 0
? lfun([bnr,[4]], s) - lfun(L4, s)
%19 ~ 0
```

The first identify identifies the nontrivial abelian character with (-47, .); the second is the factorization of the regular representation of D_{10} ; the third is the factorization of the natural representation of $D_{10} \subset S_5$; and the final four are the expressions of the degree 2 representations as induced from degree 1 representations.

lfuncheckfeq(t, precision)

Given the data attached to an *L*-function (Lmath, Ldata or Linit), check whether the functional equation is satisfied. This is most useful for an Ldata constructed "by hand", via lfuncreate, to detect mistakes.

If the function has poles, the polar part must be specified. The routine returns a bit accuracy b such that $||w - w|| < 2^b$, where w is the root number contained in data, and

$$w = \theta(1/t)t^{-k}/\overline{\theta}(t)$$

is a computed value derived from the assumed functional equation. If the parameter t is omitted, we try random samples on the real line in the segment [1, 1.25]. Of course, a large negative value of the order of realbitprecision is expected but if $\overline{\theta}$ is very small all over the sampled segment, you should first increase realbitprecision by $-\log_2 \|\overline{\theta}(t)\|$ (which is positive if θ is small) to get a meaningful result.

If t is given, it should be close to the unit disc for efficiency and such that $\overline{\theta}(t)! = 0$. We then check the functional equation at that t. Again, if $\overline{\theta}(t)$ is very small, you should first increase realbitprecision to get a useful result.

```
? \pb 128 \\ 128 bits of accuracy
? default(realbitprecision)
%1 = 128
? L = lfuncreate(1); \\ Riemann zeta
? lfuncheckfeq(L)
%3 = -124
```

i.e. the given data is consistent to within 4 bits for the particular check consisting of estimating the root number from all other given quantities. Checking away from the unit disc will either fail with a precision error, or give disappointing results (if $\theta(1/t)$ is large it will be computed with a large absolute error)

```
? lfuncheckfeq(L, 2+I)
%4 = -115
? lfuncheckfeq(L,10)
*** at top-level: lfuncheckfeq(L,10)
*** ^------
*** lfuncheckfeq: precision too low in lfuncheckfeq.
```

lfunconductor(*setN*, *flag*, *precision*)

Compute the conductor of the given *L*-function (if the structure contains a conductor, it is ignored). Two methods are available, depending on what we know about the conductor, encoded in the setN parameter:

• setN is a scalar: we know nothing but expect that the conductor lies in the interval [1, setN].

If flag is 0 (default), give either the conductor found as an integer, or a vector (possibly empty) of conductors found. If flag is 1, same but give the computed floating point approximations to the conductors found, without rounding to integers. It flag is 2, give all the conductors found, even those far from integers.

Caveat. This is a heuristic program and the result is not proven in any way:

```
? L = lfuncreate(857); \\ Dirichlet L function for kronecker(857,.)
? \p19
realprecision = 19 significant digits
? lfunconductor(L)
%2 = [17, 857]
? lfunconductor(L,,1) \\ don't round
%3 = [16.99999999999999999, 857.00000000000000]
? \p38
realprecision = 38 significant digits
? lfunconductor(L)
%4 = 857
```

Increasing setN or increasing realbitprecision slows down the program but gives better accuracy for the result. This algorithm should only be used if the primes dividing the conductor are unknown, which is uncommon.

• setN is a vector of possible conductors; for instance of the form D1 * divisors(D2), where D_1 is the known part of the conductor and D_2 is a multiple of the contribution of the bad primes.

In that case, flag is ignored and the routine uses lfuncheckfeq. It returns [N, e] where N is the best conductor in the list and e is the value of lfuncheckfeq for that N. When no suitable conductor exist or there is a tie among best potential conductors, return the empty vector [].

```
? E = ellinit([0,0,0,4,0]); /* Elliptic curve y^2 = x^3+4x */
? E.disc \\ |disc E| = 2^12
\%2 = -4096
\backslash create Ldata by hand. Guess that root number is 1 and conductor N
? L(N) = lfuncreate([n->ellan(E,n), 0, [0,1], 2, N, 1]);
\\ lfunconductor ignores conductor = 1 in Ldata !
? lfunconductor(L(1), divisors(E.disc))
\%5 = [32, -127]
? fordiv(E.disc, d, print(d,": ",lfuncheckfeq(L(d)))) \\ direct check
1: 0
2: 0
4: -1
8: -2
16: -3
32: -127
64: -3
128: -2
256: -2
512: -1
1024: -1
                                                                           (continues on next page)
```

Chapter 2. The Gen class wrapping PARI's GEN type

2048: 0 4096: 0

The above code assumed that root number was 1; had we set it to -1, none of the lfuncheckfeq values would have been acceptable:

```
? L2 = lfuncreate([n->ellan(E,n), 0, [0,1], 2, 0, -1]);
? lfunconductor(L2, divisors(E.disc))
%7 = []
```

lfuncost(sdom, der, precision)

Estimate the cost of running lfuninit(L, sdom, der) at current bit precision. Returns [t, b], to indicate that t coefficients a_n will be computed, as well as t values of gammamellininv, all at bit accuracy b. A subsequent call to lfun at s evaluates a polynomial of degree t at $\exp(hs)$ for some real parameter h, at the same bit accuracy b. If L is already an Linit, then *sdom* and *der* are ignored and are best left omitted; the bit accuracy is also inferred from L: in short we get an estimate of the cost of using that particular Linit.

```
? \pb 128
? lfuncost(1, [100]) \setminus for zeta(1/2+I*t), |t| < 100
%1 = [7, 242] \\ 7 coefficients, 242 bits
? lfuncost(1, [1/2, 100]) \\ for zeta(s) in the critical strip, |\text{Im s}| < 100
%2 = [7, 246] \\ now 246 bits
? lfuncost(1, [100], 10) \setminus for zeta(1/2+I*t), |t| < 100
\%3 = [8, 263] \setminus 10th derivative increases the cost by a small amount
? lfuncost(1, [10^5])
%3 = [158, 113438] \\ larger imaginary part: huge accuracy increase
? L = lfuncreate(polcyclo(5)); \ \ ext{Dedekind zeta for } Q(zeta_5)
? lfuncost(L, [100]) \\ at s = 1/2+I*t), |t| < 100
\%5 = [11457, 582]
? lfuncost(L, [200]) \\ twice higher
\%6 = [36294, 1035]
? lfuncost(L, [10^4]) \\ much higher: very costly !
\%7 = [70256473, 45452]
? \pb 256
? lfuncost(L, [100]); \\ doubling bit accuracy
\%8 = [17080, 710]
```

In fact, some L functions can be factorized algebraically by the lfuninit call, e.g. the Dedekind zeta function of abelian fields, leading to much faster evaluations than the above upper bounds. In that case, the function returns a vector of costs as above for each individual function in the product actually evaluated:

```
? L = lfuncreate(polcyclo(5)); \\ Dedekind zeta for Q(zeta_5)
? lfuncost(L, [100]) \\ a priori cost
%2 = [11457, 582]
? L = lfuninit(L, [100]); \\ actually perform all initializations
? lfuncost(L)
%4 = [[16, 242], [16, 242], [7, 242]]
```

The Dedekind function of this abelian quartic field is the product of four Dirichlet L-functions attached to the trivial character, a nontrivial real character and two complex conjugate characters. The nontrivial characters happen to have the same conductor (hence same evaluation costs), and correspond to two evaluations only since the two conjugate characters are evaluated simultaneously. For a total of three L-functions evaluations, which

explains the three components above. Note that the actual cost is much lower than the a priori cost in this case.

lfuncreate()

This low-level routine creates Ldata structures, needed by *lfun* functions, describing an L-function and its functional equation. We advise using a high-level constructor when one is available, see ??lfun, and this function accepts them:

```
? L = lfuncreate(1); \\ Riemann zeta
? L = lfuncreate(5); \\ Dirichlet L-function for quadratic character (5/.)
? L = lfuncreate(x^2+1); \\ Dedekind zeta for Q(i)
? L = lfuncreate(ellinit([0,1])); \\ L-function of E/Q: y^2=x^3+1
```

One can then use, e.g., lfun(L, s) to directly evaluate the respective L-functions at s, or lfuninit(L, [c,w,h]) to initialize computations in the rectangular box $\Re(s-c) \le w$, $\Im(s) \le h$.

We now describe the low-level interface, used to input nonbuiltin L-functions. The input is now a 6 or 7 component vector V = [a, astar, Vga, k, N, eps, poles], whose components are as follows:

V[1] = a encodes the Dirichlet series coefficients (a_n). The preferred format is a closure of arity 1: n- > vector(n,i,a(i)) giving the vector of the first n coefficients. The closure is allowed to return a vector of more than n coefficients (only the first n will be considered) or even less than n, in which case loss of accuracy will occur and a warning that #an is less than expected is issued. This allows to precompute and store a fixed large number of Dirichlet coefficients in a vector v and use the closure n- > v, which does not depend on n. As a shorthand for this latter case, you can input the vector v itself instead of the closure.

```
? z = lfuncreate([n->vector(n,i,1), 1, [0], 1, 1, 1, 1]); \\ Riemann zeta
? lfun(z,2) - Pi^2/6
%2 = -5.877471754111437540 E-39
```

A second format is limited to L-functions affording an Euler product. It is a closure of arity 2 (p,d)- > F(p) giving the local factor $L_p(X)$ at p as a rational function, to be evaluated at p^{-s} as in direuler; d is set to logint(n, p) + 1, where n is the total number of Dirichlet coefficients $(a_1, ..., a_n)$ that will be computed. In other words, the smallest integer d such that $p^d > n$. This parameter d allows to compute only part of L_p when p is large and L_p expensive to compute: any polynomial (or t_SER) congruent to L_p modulo X^d is acceptable since only the coefficients of $X^0, ..., X^{d-1}$ are needed to expand the Dirichlet series. The closure can of course ignore this parameter:

```
? z = lfuncreate([(p,d)->1/(1-x), 1, [0], 1, 1, 1, 1]); \\ Riemann zeta
? lfun(z,2) - Pi^2/6
%4 = -5.877471754111437540 E-39
```

One can describe separately the generic local factors coefficients and the bad local factors by setting $dir = [F, L_{bad}]$, were $L_{bad} = [[p_1, L_{p_1}], ..., [p_k, L_{p_k}]]$, where F describes the generic local factors as above, except that when $p = p_i$ for some i <= k, the coefficient a_p is directly set to L_{p_i} instead of calling F.

```
N = 15;
E = ellinit([1, 1, 1, -10, -10]); \\ = "15a1"
F(p,d) = 1 / (1 - ellap(E,p)*'x + p*'x^2);
Lbad = [[3, 1/(1+'x)], [5, 1/(1-'x)]];
L = lfuncreate([[F,Lbad], 0, [0,1], 2, N, ellrootno(E)]);
```

Of course, in this case, lfuncreate(E) is preferable!

- V[2] = astar is the Dirichlet series coefficients of the dual function, encoded as a above. The sentinel values 0 and 1 may be used for the special cases where $a = a^*$ and $a = \overline{a^*}$, respectively.
- V[3] = Vga is the vector of α_j such that the gamma factor of the L-function is equal to

where : math : $\Gamma_{\mathbb{R}}(s) = \pi^{-s/2} \Gamma(s/2)$. This same syntaxis used in the : literal : 'gamma mellinin' functions. In particular the set of the set o

• V[4] = k is a positive integer k. The functional equation relates values at s and k - s. For instance, for an Artin L-series such as a Dedekind zeta function we have k = 1, for an elliptic curve k = 2, and for a modular form, k is its weight. For motivic L-functions, the *motivic* weight w is w = k - 1.

By default we assume that $a_n = O_{\epsilon}(n^{k_1+\epsilon})$, where $k_1 = w$ and even $k_1 = w/2$ when the L function has no pole (Ramanujan-Petersson). If this is not the case, you can replace the k argument by a vector $[k, k_1]$, where k_1 is the upper bound you can assume.

- V[5] = N is the conductor, an integer $N \ge 1$, such that $\Lambda(s) = N^{s/2} \gamma_A(s) L(s)$ with $\gamma_A(s)$ as above.
- V[6] = eps is the root number ε , i.e., the complex number (usually of modulus 1) such that $\Lambda(a, k s) = \varepsilon \Lambda(a^*, s)$.
- The last optional component V[7] = poles encodes the poles of the L or Λ-functions, and is omitted if they have no poles. A polar part is given by a list of 2-component vectors [β, P_β(x)], where β is a pole and the power series P_β(x) describes the attached polar part, such that L(s) P_β(s β) is holomorphic in a neighbourhood of β. For instance P_β = r/x + O(1) for a simple pole at β or r₁/x² + r₂/x + O(1) for a double pole. The type of the list describing the polar part allows to distinguish between L and Λ: a t_VEC is attached to L, and a t_COL is attached to Λ. Unless a = a^{*} (coded by astar equal to 0 or 1), it is mandatory to specify the polar part of Λ rather than those of L since the poles of L* cannot be infered from the latter ! Whereas the functional equation allows to deduce the polar part of Λ* from the polar part of Λ.

Finally, if $a = \overline{a^*}$, we allow a shortcut to describe the frequent situation where L has at most simple pole, at s = k, with residue r a complex scalar: you may then input poles = r. This value r can be set to 0 if unknown and it will be computed.

When one component is not exact. Alternatively, obj can be a closure of arity 0 returning the above vector to the current real precision. This is needed if some components are not available exactly but only through floating point approximations. The closure allows algorithms to recompute them to higher accuracy when needed. Compare

```
? Ld1() = [n->lfunan(Mod(2,7),n),1,[0],1,7,((-13-3*sqrt(-3))/14)^(1/6)];
? Ld2 = [n->lfunan(Mod(2,7),n),1,[0],1,7,((-13-3*sqrt(-3))/14)^(1/6)];
? L1 = lfuncreate(Ld1);
? L2 = lfuncreate(Ld2);
? lfun(L1,1/2+I*200) \\ OK
%5 = 0.55943925130316677665287870224047183265 -
0.42492662223174071305478563967365980756*I
? lfun(L2,1/2+I*200) \\ all accuracy lost
%6 = 0.E-38 + 0.E-38*I
```

The accuracy lost in Ld2 is due to the root number being given to an insufficient precision. To see what happens try

```
? Ld3() = printf("prec needed: %ld bits",getlocalbitprec());Ld1()
? L3 = lfuncreate(Ld3);
prec needed: 64 bits
? z3 = lfun(L3,1/2+I*200)
prec needed: 384 bits
%16 = 0.55943925130316677665287870224047183265 -
0.42492662223174071305478563967365980756*I
```

lfundiv(L2, precision)

Creates the Ldata structure (without initialization) corresponding to the quotient of the Dirichlet series L_1 and

 L_2 given by L1 and L2. Assume that $v_z(L_1) \ge v_z(L_2)$ at all complex numbers z: the construction may not create new poles, nor increase the order of existing ones.

lfundual(precision)

Creates the Ldata structure (without initialization) corresponding to the dual L-function ^L of L. If k and ε are respectively the weight and root number of L, then the following formula holds outside poles, up to numerical errors:

$$\Lambda(L,s) = \varepsilon \Lambda(^L, k-s).$$

? L = lfunqf(matdiagonal([1,2,3,4])); ? eps = lfunrootres(L)[3]; k = L[4]; ? M = lfundual(L); lfuncheckfeq(M) %3 = -127 ? s= 1+Pi*I; ? a = lfunlambda(L,s); ? b = eps * lfunlambda(M,k-s); ? exponent(a - b) %7 = -130

lfunetaquo()

Returns the Ldata structure attached to the L function attached to the modular form $z : - - - > \prod_{i=1}^{n} \eta(M_{i,1}z)^{M_{i,2}}$ It is currently assumed that f is a self-dual cuspidal form on $\Gamma_0(N)$ for some N. For instance, the L-function $\sum \tau(n)n^{-s}$ attached to Ramanujan's Δ function is encoded as follows

? L = lfunetaquo(Mat([1,24])); ? lfunan(L, 100) \\ first 100 values of tau(n)

For convenience, a t_VEC is also accepted instead of a factorization matrix with a single row:

```
? L = lfunetaquo([1,24]); \\ same as above
```

lfungenus2()

Returns the Ldata structure attached to the L function attached to the genus-2 curve defined by $y^2 = F(x)$ or $y^2 + Q(x)y = P(x)$ if F = [P,Q]. Currently, the model needs to be minimal at 2, and if the conductor is even, its valuation at 2 might be incorrect (a warning is issued).

lfunhardy(*t*, *precision*)

Variant of the Hardy Z-function given by L, used for plotting or locating zeros of L(k/2+it) on the critical line. The precise definition is as follows: let k/2 be the center of the critical strip, d be the degree, $Vga = (\alpha_j)_{j < =d}$ given the gamma factors, and ε be the root number; we set $s = k/2 + it = \rho e^{i\theta}$ and $2E = d(k/2 - 1) + \Re(\sum_{1 < =j < =d} \alpha_j)$. Assume first that Λ is self-dual, then the computed function at t is equal to

$$Z(t) = \varepsilon^{-1/2} \Lambda(s) . \rho^{-E} e^{dt\theta/2}$$

which is a real function of t vanishing exactly when L(k/2 + it) does on the critical line. The normalizing factor $||s||^{-E}e^{dt\theta/2}$ compensates the exponential decrease of $\gamma_A(s)$ as $t \to oo$ so that Z(t) 1. For non-self-dual Λ , the definition is the same except we drop the $\varepsilon^{-1/2}$ term (which is not well defined since it depends on the chosen dual sequence $a^*(n)$): Z(t) is still of the order of 1 and still vanishes where L(k/2 + it) does, but it needs no longer be real-valued.

? T = 100; \\ maximal height
? L = lfuninit(1, [T]); \\ initialize for zeta(1/2+it), |t|<T
? \p19 \\ no need for large accuracy
? ploth(t = 0, T, lfunhardy(L,t))</pre>

Using lfuninit is critical for this particular applications since thousands of values are computed. Make sure to initialize up to the maximal t needed: otherwise expect to see many warnings for unsufficient initialization and suffer major slowdowns.

lfuninit(*sdom*, *der*, *precision*)

Initialization function for all functions linked to the computation of the *L*-function L(s) encoded by L, where *s* belongs to the rectangular domain sdom = [center, w, h] centered on the real axis, $||\Re(s) - center|| \le w$, $||\Im(s)|| \le h$, where all three components of sdom are real and *w*, *h* are nonnegative. der is the maximum order of derivation that will be used. The subdomain [k/2, 0, h] on the critical line (up to height *h*) can be encoded as [h] for brevity. The subdomain [k/2, w, h] centered on the critical line can be encoded as [w, h] for brevity.

The argument L is an Lmath, an Ldata or an Linit. See ??Ldata and ??lfuncreate for how to create it.

The height h of the domain is a *crucial* parameter: if you only need L(s) for real s, set h to 0. The running time is roughly proportional to

$$(B/d + \pi h/4)^{d/2 + 3} N^{1/2}$$

where B is the default bit accuracy, d is the degree of the L-function, and N is the conductor (the exponent d/2+3 is reduced to d/2 + 2 when d = 1 and d = 2). There is also a dependency on w, which is less crucial, but make sure to use the smallest rectangular domain that you need.

```
? L0 = lfuncreate(1); \\ Riemann zeta
? L = lfuninit(L0, [1/2, 0, 100]); \\ for zeta(1/2+it), |t| < 100
? lfun(L, 1/2 + I)
? L = lfuninit(L0, [100]); \\ same as above !
```

lfunlambda(s, D, precision)

Compute the completed *L*-function $\Lambda(s) = N^{s/2}\gamma(s)L(s)$, or if D is set, the derivative of order D at *s*. The parameter L is either an Lmath, an Ldata (created by lfuncreate, or an Linit (created by lfuninit), preferrably the latter if many values are to be computed.

The result is given with absolute error less than $2^{-B} \|\gamma(s) N^{s/2}\|$, where B = real bit precision.

lfunmf(*F*, *precision*)

If F is a modular form in mf, output the L-functions corresponding to its $[\mathbb{Q}(F) : \mathbb{Q}(\chi)]$ complex embeddings, ready for use with the lfun package. If F is omitted, output the L-functions attached to all eigenforms in the new space; the result is a vector whose length is the number of Galois orbits of newforms. Each entry contains the vector of L-functions corresponding to the d complex embeddings of an orbit of dimension d over $\mathbb{Q}(\chi)$.

```
? mf = mfinit([35,2],0);mffields(mf)
%1 = [y, y^2 - y - 4]
? f = mfeigenbasis(mf)[2]; mfparams(f) \\ orbit of dimension two
%2 = [35, 2, 1, y^2 - y - 4, t - 1]
? [L1,L2] = lfunmf(mf, f); \\ Two L-functions
? lfun(L1,1)
%4 = 0.81018461849460161754947375433874745585
? lfun(L2,1)
%5 = 0.46007635204895314548435893464149369804
? [ lfun(L,1) | L <- concat(lfunmf(mf)) ]
%6 = [0.70291..., 0.81018..., 0.46007...]
```

The concat instruction concatenates the vectors corresponding to the various (here two) orbits, so that we obtain the vector of all the *L*-functions attached to eigenforms.

lfunmfspec(precision)

Let L be the L-function attached to a modular eigenform f of weight k, as given by lfunmf. In even weight, returns [ve,vo,om,op], where ve (resp., vo) is the vector of even (resp., odd) periods of f and om and op the corresponding real numbers ω^- and ω^+ normalized in a noncanonical way. In odd weight ominus is the same as op and we return [v,op] where v is the vector of all periods.

```
? D = mfDelta(); mf = mfinit(D); L = lfunmf(mf, D);
? [ve, vo, om, op] = lfunmfspec(L)
%2 = [[1, 25/48, 5/12, 25/48, 1], [1620/691, 1, 9/14, 9/14, 1, 1620/691],\
0.0074154209298961305890064277459002287248,\
0.0050835121083932868604942901374387473226]
? DS = mfsymbol(mf, D); bestappr(om*op / mfpetersson(DS), 10^8)
%3 = 8192/225
? mf = mfinit([4, 9, -4], 0);
? F = mfeigenbasis(mf)[1]; L = lfunmf(mf, F);
? [v, om] = lfunmfspec(L)
%6 = [[1, 10/21, 5/18, 5/24, 5/24, 5/18, 10/21, 1],\
1.1302643192034974852387822584241400608]
? FS = mfsymbol(mf, F); bestappr(om^2 / mfpetersson(FS), 10^8)
%7 = 113246208/325
```

lfunmul(*L2*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the product of the Dirichlet series given by L1 and L2.

lfunorderzero(*m*, *precision*)

Computes the order of the possible zero of the L-function at the center k/2 of the critical strip; return 0 if L(k/2) does not vanish.

If m is given and has a nonnegative value, assumes the order is at most m. Otherwise, the algorithm chooses a sensible default:

- if the L argument is an Linit, assume that a multiple zero at s = k/2 has order less than or equal to the maximal allowed derivation order.
- else assume the order is less than 4.

You may explicitly increase this value using optional argument *m*; this overrides the default value above. (Possibly forcing a recomputation of the Linit.)

lfunqf(precision)

Returns the Ldata structure attached to the Θ function of the lattice attached to the primitive form proportional to the definite positive quadratic form Q.

? L = lfunqf(matid(2)); ? lfunqf(L,2) %2 = 6.0268120396919401235462601927282855839 ? lfun(x^2+1,2)*4 %3 = 6.0268120396919401235462601927282855839

The following computes the Madelung constant:

```
? L1=lfunqf(matdiagonal([1,1,1]));
? L2=lfunqf(matdiagonal([4,1,1]));
? L3=lfunqf(matdiagonal([4,4,1]));
? F(s)=6*lfun(L2,s)-12*lfun(L3,s)-lfun(L1,s)*(1-8/4^s);
```

```
? F(1/2)
%5 = -1.7475645946331821906362120355443974035
```

lfunrootres(precision)

Given the Ldata attached to an *L*-function (or the output of lfunthetainit), compute the root number and the residues.

The output is a 3-component vector $[[[a_1, r_1], ..., [a_n, r_n], [[b_1, R_1], ..., [b_m, R_m]], w]$, where r_i is the polar part of L(s) at a_i , R_i is the polar part of $\Lambda(s)$ at b_i or [0, 0, r] if there is no pole, and w is the root number. In the present implementation,

• either the polar part must be completely known (and is then arbitrary): the function determines the root number,

```
? L = lfunmul(1,1); \\ zeta^2
? [r,R,w] = lfunrootres(L);
? r \\ single pole at 1, double
%3 = [[1, 1.[...]*x^-2 + 1.1544[...]*x^-1 + O(x^0)]]
? w
%4 = 1
? R \\ double pole at 0 and 1
%5 = [[1,[...]], [0,[...]]]~
```

• or at most a single pole is allowed: the function computes both the root number and the residue (0 if no pole).

lfunshift(*d*, *flag*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the shift of L by d, that is to the function L_d such that $L_d(s) = L(s - d)$. If flag = 1, return the product LxL_d instead.

lfunsympow(*m*)

Returns the Ldata structure attached to the L function attached to the m-th symmetric power of the elliptic curve E defined over the rationals.

lfuntheta(*t*, *m*, *precision*)

Compute the value of the *m*-th derivative at *t* of the theta function attached to the *L*-function given by data. data can be either the standard *L*-function data, or the output of lfunthetainit. The result is given with absolute error less than 2^{-B} , where B = real bit precision.

The theta function is defined by the formula $\Theta(t) = \sum_{n>=1} a(n)K(nt/\sqrt{N})$, where a(n) are the coefficients of the Dirichlet series, N is the conductor, and K is the inverse Mellin transform of the gamma product defined by the Vga component. Its Mellin transform is equal to $\Lambda(s) - P(s)$, where $\Lambda(s)$ is the completed L-function and the rational function P(s) its polar part. In particular, if the L-function is the L-function of a modular form

 $f(\tau) = \sum_{n>=0} a(n)q^n$ with $q = \exp(2\pi i \tau)$, we have $\Theta(t) = 2(f(it/\sqrt{N}) - a(0))$. Note that a(0) = -L(f, 0) in this case.

lfunthetacost(tdom, m, precision)

This function estimates the cost of running lfunthetainit(L,tdom,m) at current bit precision. Returns the number of coefficients a_n that would be computed. This also estimates the cost of a subsequent evaluation lfuntheta, which must compute that many values of gammamellininv at the current bit precision. If L is already an Linit, then *tdom* and m are ignored and are best left omitted: we get an estimate of the cost of using that particular Linit.

```
? \pb 1000
? L = lfuncreate(1); \\ Riemann zeta
? lfunthetacost(L); \setminus cost for theta(t), t real >= 1
\%1 = 15
? lfunthetacost(L, 1 + I); \\ cost for theta(1+I). Domain error !
*** at top-level: lfunthetacost(1,1+I)
*** ^_____
 *** lfunthetacost: domain error in lfunthetaneed: arg t > 0.785
? lfunthetacost(L, 1 + I/2) \\ for theta(1+I/2).
\%2 = 23
? lfunthetacost(L, 1 + I/2, 10) \\ for theta^((10))(1+I/2).
\%3 = 24
? lfunthetacost(L, [2, 1/10]) \setminus cost for theta(t), |t| \ge 2, |arg(t)| < 1/10
%4 = 8
? L = lfuncreate( ellinit([1,1]) );
? lfunthetacost(L) \setminus for t >= 1
\%6 = 2471
```

lfunthetainit(tdom, m, precision)

Initialization function for evaluating the *m*-th derivative of theta functions with argument t in domain tdom. By default (tdom omitted), t is real, $t \ge 1$. Otherwise, tdom may be

- a positive real scalar ρ : t is real, $t \ge \rho$.
- a nonreal complex number: compute at this particular t; this allows to compute θ(z) for any complex z satisfying ||z|| >= ||t|| and || arg z|| <= || arg t||; we must have ||2 arg z/d|| < π/2, where d is the degree of the Γ factor.
- a pair [ρ, α]: assume that ||t|| >= ρ and || arg t|| ≤ α; we must have ||2α/d|| < π/2, where d is the degree of the Γ factor.

```
? \p500
? L = lfuncreate(1); \\ Riemann zeta
? t = 1+I/2;
? lfuntheta(L, t); \\ direct computation
time = 30 ms.
? T = lfunthetainit(L, 1+I/2);
time = 30 ms.
? lfuntheta(T, t); \\ instantaneous
```

The T structure would allow to quickly compute $\theta(z)$ for any z in the cone delimited by t as explained above. On the other hand

```
? lfuntheta(T,I)
 *** at top-level: lfuntheta(T,I)
 *** ^------
 *** lfuntheta: domain error in lfunthetaneed: arg t > 0.785398163397448
```

The initialization is equivalent to

```
? lfunthetainit(L, [abs(t), arg(t)])
```

lfuntwist(*chi*, *precision*)

Creates the Ldata structure (without initialization) corresponding to the twist of L by the primitive character attached to the Dirichlet character chi. The conductor of the character must be coprime to the conductor of the L-function L.

lfunzeros(lim, divz, precision)

lim being either a positive upper limit or a nonempty real interval, computes an ordered list of zeros of L(s) on the critical line up to the given upper limit or in the given interval. Use a naive algorithm which may miss some zeros: it assumes that two consecutive zeros at height $T \ge 1$ differ at least by $2\pi/\omega$, where

 $\omega := divz.(d\log(T/2\pi) + d + 2\log(N/(\pi/2)^d)).$

To use a finer search mesh, set divz to some integral value larger than the default (= 8).

```
? lfunzeros(1, 30) \\ zeros of Rieman zeta up to height 30
%1 = [14.134[...], 21.022[...], 25.010[...]]
? #lfunzeros(1, [100,110]) \\ count zeros with 100 <= Im(s) <= 110
%2 = 4</pre>
```

The algorithm also assumes that all zeros are simple except possibly on the real axis at s = k/2 and that there are no poles in the search interval. (The possible zero at s = k/2 is repeated according to its multiplicity.)

If you pass an Linit to the function, the algorithm assumes that a multiple zero at s = k/2 has order less than or equal to the maximal derivation order allowed by the Linit. You may increase that value in the Linit but this is costly: only do it for zeros of low height or in lfunorderzero instead.

lift(v)

If v is omitted, lifts intmods from $\mathbb{Z}/n\mathbb{Z}$ in \mathbb{Z} , p-adics from \mathbb{Q}_p to \mathbb{Q} (as truncate), and polmods to polynomials. Otherwise, lifts only polmods whose modulus has main variable v. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(lift,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

```
? lift(Mod(5,3))
%1 = 2
? lift(3 + 0(3^9))
%2 = 3
? lift(Mod(x,x^2+1))
%3 = x
? lift(Mod(x,x^2+1))
%4 = x
```

Lifts are performed recursively on an object components, but only by *one level*: once a t_POLMOD is lifted, the components of the result are *not* lifted further.

```
? lift(x * Mod(1,3) + Mod(2,3))
%4 = x + 2
? lift(x * Mod(y,y^2+1) + Mod(2,3))
%5 = y*x + Mod(2, 3) \\ do you understand this one?
? lift(x * Mod(y,y^2+1) + Mod(2,3), 'x)
%6 = Mod(y, y^2 + 1)*x + Mod(Mod(2, 3), y^2 + 1)
? lift(%, y)
%7 = y*x + Mod(2, 3)
```

To recursively lift all components not only by one level, but as long as possible, use liftall. To lift only t_INTMOD s and t_PADIC s components, use liftint. To lift only t_POLMOD s components, use liftpol. Finally, centerlift allows to lift t_INTMOD s and t_PADIC s using centered residues (lift of smallest absolute value).

liftall()

Recursively lift all components of x from $\mathbb{Z}/n\mathbb{Z}$ to \mathbb{Z} , from \mathbb{Q}_p to \mathbb{Q} (as truncate), and polmods to polynomials. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftall, L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

? liftall(x * (1 + 0(3)) + Mod(2,3))
%1 = x + 2
? liftall(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2))
%2 = y*x + 2*z

liftint()

Recursively lift all components of x from $\mathbb{Z}/n\mathbb{Z}$ to \mathbb{Z} and from \mathbb{Q}_p to \mathbb{Q} (as truncate). t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftint,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

```
? liftint(x * (1 + 0(3)) + Mod(2,3))
%1 = x + 2
? liftint(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2))
%2 = Mod(y, y^2 + 1)*x + Mod(Mod(2*z, z^2), y^2 + 1)
```

liftpol()

Recursively lift all components of x which are polmods to polynomials. t_FFELT are not lifted, nor are List elements: you may convert the latter to vectors first, or use apply(liftpol,L). More generally, components for which such lifts are meaningless (e.g. character strings) are copied verbatim.

```
? liftpol(x * (1 + 0(3)) + Mod(2,3))
%1 = (1 + 0(3))*x + Mod(2, 3)
? liftpol(x * Mod(y,y^2+1) + Mod(2,3)*Mod(z,z^2))
%2 = y*x + Mod(2, 3)*z
```

limitnum(alpha, precision)

Lagrange-Zagier numerical extrapolation of *expr*, corresponding to a sequence u_n , either given by a closure n-> u(n). I.e., assuming that u_n tends to a finite limit ℓ , try to determine ℓ .

The routine assume that u_n has an asymptotic expansion in $n^{-\alpha}$:

$$u_n = \ell + \sum_{i>=1} a_i n^{-i\alpha}$$

for some a_i . It is purely numerical and heuristic, thus may or may not work on your examples. The expression will be evaluated for n = 1, 2, ..., N for an N = O(B) at a bit accuracy bounded by 1.612B.

It is not mandatory to specify α when the u_n have an asymptotic expansion in n^{-1} . However, if the series in n^{-1} is lacunary, specifying α allows faster computation:

```
? \p1000
? limitnum(n->(1+1/n^2)^(n^2)) - exp(1)
time = 1min, 44,681 ms.
%4 = 0.E-1001
? limitnum(n->(1+1/n^2)^(n^2), 2) - exp(1)
time = 27,271 ms.
%5 = 0.E-1001 \\ still perfect, 4 times faster
```

When u_n has an asymptotic expansion in $n^{-\alpha}$ with α not an integer, leaving α unspecified will bring an inexact limit. Giving a satisfying optional argument improves precision; the program runs faster when the optional argument gives non lacunary series.

```
? \p50
? limitnum(n->(1+1/n^(7/2))^(n^(7/2))) - exp(1)
time = 982 ms.
%6 = 4.13[...] E-12
? limitnum(n->(1+1/n^(7/2))^(n^(7/2)), 1/2) - exp(1)
time = 16,745 ms.
%7 = 0.E-57
? limitnum(n->(1+1/n^(7/2))^(n^(7/2)), 7/2) - exp(1)
time = 105 ms.
%8 = 0.E-57
```

Alternatively, u_n may be given by a closure $N : -- > [u_1, ..., u_N]$ which can often be programmed in a more efficient way, for instance when u_{n+1} is a simple function of the preceding terms:

```
? \p2000
? limitnum(n -> 2^(4*n+1)*(n!)^4 / (2*n)! /(2*n+1)! ) - Pi
time = 1,755 ms.
%9 = 0.E-2003
? vu(N) = \\ exploit hypergeometric property
{ my(v = vector(N)); v[1] = 8./3;\
for (n=2, N, my(q = 4*n^2); v[n] = v[n-1]*q/(q-1));\
return(v);
}
? limitnum(vu) - Pi \\ much faster
time = 106 ms.
%11 = 0.E-2003
```

All sums and recursions can be handled in the same way. In the above it is essential that u_n be defined as a closure because it must be evaluated at a higher precision than the one expected for the limit. Make sure that the closure does not depend on a global variable which would be computed at a priori fixed accuracy. For instance,

precomputing v1 = 8.0/3 first and using v1 in vu above would be wrong because the resulting vector of values will use the accuracy of v1 instead of the ambient accuracy at which limitnum will call it.

Alternatively, and more clumsily, u_n may be given by a vector of values: it must be long and precise enough for the extrapolation to make sense. Let B be the current realbitprecision, the vector length must be at least 1.102B and the values computed with bit accuracy 1.612B.

```
? limitnum(vector(10,n,(1+1/n)^n))
 *** ^_____
 *** limitnum: nonexistent component in limitnum: index < 43
\\ at this accuracy, we must have at least 43 values
? limitnum(vector(43,n,(1+1/n)^n)) - exp(1)
\%12 = 0.E - 37
? v = vector(43);
? s = 0; for(i=1,#v, s += 1/i; v[i]= s - log(i));
? limitnum(v) - Euler
%15 = -1.57[...] E-16
? v = vector(43);
\\ ~ 128 bit * 1.612
? localbitprec(207);\
s = 0; for(i=1,#v, s += 1/i; v[i] = s - log(i));
? limitnum(v) - Euler
\%18 = 0.E - 38
```

Because of the above problems, the preferred format is thus a closure, given either a single value or the vector of values $[u_1, ..., u_N]$. The function distinguishes between the two formats by evaluating the closure at N! = 1 and 1 and checking whether it yields vectors of respective length N and 1 or not.

Warning. The expression is evaluated for n = 1, 2, ..., N for an N = O(B) if the current bit accuracy is B. If it is not defined for one of these values, translate or rescale accordingly:

```
? limitnum(n->log(1-1/n)) \\ can't evaluate at n = 1 !
    *** at top-level: limitnum(n->log(1-1/n))
    *** ^-------
    *** in function limitnum: log(1-1/n)
    *** ^------
    *** log: domain error in log: argument = 0
? limitnum(n->-log(1-1/(2*n)))
%19 = -6.11[...] E-58
```

We conclude with a complicated example. Since the function is heuristic, it is advisable to check whether it produces the same limit for $u_n, u_{2n}, ..., u_{km}$ for a suitable small multiplier k. The following function implements the recursion for the Motzkin numbers M_n which count the number of ways to draw non intersecting chords between n points on a circle:

$$M_n = M_{n-1} + \sum_{i < n-1} M_i M_{n-2-i} = ((n+1)M_{n-1} + (3n-3)M_{n-2})/(n+2).$$

It is known that $M_n (3^{n+1})/(\sqrt{12\pi n^3})$.

```
\\ [M_k, M_{k^2}, ..., M_{k^N}] / (3^n / n^(3/2))
vM(N, k = 1) =
{ my(q = k*N, V);
```

```
if (q == 1, return ([1/3]));
V = vector(q); V[1] = V[2] = 1;
 for (n = 2, q - 1),
V[n+1] = ((2*n + 1)*V[n] + 3*(n - 1)*V[n-1]) / (n + 2));
 f = (n \rightarrow 3^n / n^3/2);
return (vector(N, n, V[n*k] / f(n*k));
}
? limitnum(vM) - 3/sqrt(12*Pi) \\ complete junk
%1 = 35540390.753542730306762369615276452646
? limitnum(N->vM(N,5)) - 3/sqrt(12*Pi) \\ M_{5n}: better
%2 = 4.130710262178469860 E-25
? limitnum(N->vM(N,10)) - 3/sqrt(12*Pi) \\ M_{10n}: perfect
\%3 = 0.E - 38
? \p2000
? limitnum(N->vM(N,10)) - 3/sqrt(12*Pi) \setminus also at high accuracy
time = 409 \text{ ms}.
\%4 = 1.1048895470044788191 E-2004
```

In difficult cases such as the above a multiplier of 5 to 10 is usually sufficient. The above example is typical: a good multiplier usually remains sufficient when the requested precision increases!

lindep(flag)

finds a small nontrivial integral linear combination between components of v. If none can be found return an empty vector.

If v is a vector with real/complex entries we use a floating point (variable precision) LLL algorithm. If flag = 0 the accuracy is chosen internally using a crude heuristic. If flag > 0 the computation is done with an accuracy of flag decimal digits. To get meaningful results in the latter case, the parameter flag should be smaller than the number of correct decimal digits in the input.

? lindep([sqrt(2), sqrt(3), sqrt(2)+sqrt(3)])
%1 = [-1, -1, 1]~

If v is p-adic, flag is ignored and the algorithm LLL-reduces a suitable (dual) lattice.

```
? lindep([1, 2 + 3 + 3<sup>2</sup> + 3<sup>3</sup> + 3<sup>4</sup> + 0(3<sup>5</sup>)])
%2 = [1, -2]~
```

If v is a matrix (or a vector of column vectors, or a vector of row vectors), flag is ignored and the function returns a non trivial kernel vector if one exists, else an empty vector.

```
? lindep([1,2,3;4,5,6;7,8,9])
%3 = [1, -2, 1]~
? lindep([[1,0], [2,0]])
%4 = [2, -1]~
? lindep([[1,0], [0,1]])
%5 = []~
```

If v contains polynomials or power series over some base field, finds a linear relation with coefficients in the field.

```
? lindep([x*y, x^2 + y, x^2*y + x*y^2, 1])
%4 = [y, y, -1, -y^2]~
```

For better control, it is preferable to use t_POL rather than t_SER in the input, otherwise one gets a linear combination which is *t*-adically small, but not necessarily 0. Indeed, power series are first converted to the minimal absolute accuracy occurring among the entries of v (which can cause some coefficients to be ignored), then truncated to polynomials:

? v = [t²+0(t⁴), 1+0(t²)]; L=lindep(v)
%1 = [1, 0]~
? v*L
%2 = t²+0(t⁴) \\ small but not 0

listinsert(n, _arg2)

Inserts the object x at position n in L (which must be of type t_LIST). This has complexity O(#L - n + 1): all the remaining elements of *list* (from position n + 1 onwards) are shifted to the right. If n is greater than the list length, appends x.

```
? L = List([1,2,3]);
? listput(~L, 4); L \\ listput inserts at end
%4 = List([1, 2, 3, 4])
? listinsert(~L, 5, 1); L \\insert at position 1
%5 = List([5, 1, 2, 3, 4])
? listinsert(~L, 6, 1000); L \\ trying to insert beyond position #L
%6 = List([5, 1, 2, 3, 4, 6]) \\ ... inserts at the end
```

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

listkill()

Obsolete, retained for backward compatibility. Just use L = List() instead of listkill(L). In most cases, you won't even need that, e.g. local variables are automatically cleared when a user function returns.

listpop(_arg1)

Removes the *n*-th element of the list *list* (which must be of type t_LIST). If *n* is omitted, or greater than the list current length, removes the last element. If the list is already empty, do nothing. This runs in time O(#L-n+1).

? L = List([1,2,3,4]); ? listpop(~L); L \\ remove last entry %2 = List([1, 2, 3]) ? listpop(~L, 1); L \\ remove first entry %3 = List([2, 3])

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

listput(n, _arg2)

Sets the *n*-th element of the list *list* (which must be of type t_LIST) equal to x. If n is omitted, or greater than the list length, appends x. The function returns the inserted element.

```
? L = List();
? listput(~L, 1)
%2 = 1
? listput(~L, 2)
%3 = 2
? L
%4 = List([1, 2])
```

Note the ~ L: this means that the function is called with a *reference* to L and changes L in place.

You may put an element into an occupied cell (not changing the list length), but it is easier to use the standard list[n] = x construct.

```
? listput(~L, 3, 1) \\ insert at position 1
%5 = 3
? L
%6 = List([3, 2])
? L[2] = 4 \\ simpler
%7 = List([3, 4])
? L[10] = 1 \\ can't insert beyond the end of the list
 *** at top-level: L[10]=1
 *** ^-----
 *** nonexistent component: index > 2
? listput(L, 1, 10) \\ but listput can
%8 = 1
? L
%9 = List([3, 2, 1])
```

This function runs in time O(#L) in the worst case (when the list must be reallocated), but in time O(1) on average: any number of successive listput s run in time O(#L), where #L denotes the list *final* length.

listsort(_arg1)

Sorts the t_LIST *list* in place, with respect to the (somewhat arbitrary) universal comparison function cmp. In particular, the ordering is the same as for sets and setsearch can be used on a sorted list. No value is returned. If flag is nonzero, suppresses all repeated coefficients.

```
? L = List([1,2,4,1,3,-1]); listsort(~L); L
%1 = List([-1, 1, 1, 2, 3, 4])
? setsearch(L, 4)
%2 = 6
? setsearch(L, -2)
%3 = 0
? listsort(~L, 1); L \\ remove duplicates
%4 = List([-1, 1, 2, 3, 4])
```

Note the \sim L: this means that the function is called with a *reference* to L and changes L in place: this is faster than the vecsort command since the list is sorted in place and we avoid unnecessary copies.

```
? v = vector(100,i,random); L = List(v);
? for(i=1,10^4, vecsort(v))
time = 162 ms.
? for(i=1,10^4, vecsort(L))
time = 162 ms.
? for(i=1,10^4, listsort(~L))
time = 63 ms.
```

lngamma(precision)

Principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with nonpositive integers removed, and can have much larger arguments than gamma itself.

For x a power series such that x(0) is not a pole of gamma, compute the Taylor expansion. (PARI only knows about regular power series and can't include logarithmic terms.)

```
? lngamma(1+x+0(x^2))
%1 = -0.57721566490153286060651209008240243104*x + 0(x^2)
? lngamma(x+0(x^2))
*** at top-level: lngamma(x+0(x^2))
```

```
*** ^-----
*** Ingamma: domain error in lngamma: valuation != 0
? lngamma(-1+x+0(x^2))
*** Ingamma: Warning: normalizing a series with 0 leading term.
*** at top-level: lngamma(-1+x+0(x^2))
*** ^-----
*** lngamma: domain error in intformal: residue(series, pole) != 0
```

localbitprec()

Set the real precision to p bits in the dynamic scope. All computations are performed as if realbitprecision was p: transcendental constants (e.g. Pi) and conversions from exact to floating point inexact data use p bits, as well as iterative routines implicitly using a floating point accuracy as a termination criterion (e.g. solve or intnum). But realbitprecision itself is unaffected and is "unmasked" when we exit the dynamic (*not* lexical) scope. In effect, this is similar to

```
my(bit = default(realbitprecision));
default(realbitprecision,p);
....
default(realbitprecision, bit);
```

but is both less cumbersome, cleaner (no need to manipulate a global variable, which in fact never changes and is only temporarily masked) and more robust: if the above computation is interrupted or an exception occurs, realbitprecision will not be restored as intended.

Such localbitprec statements can be nested, the innermost one taking precedence as expected. Beware that localbitprec follows the semantic of local, not my: a subroutine called from localbitprec scope uses the local accuracy:

```
? f()=bitprecision(1.0);
? f()
%2 = 128
? localbitprec(1000); f()
%3 = 1024
```

Note that the bit precision of *data* (1.0 in the above example) increases by steps of 64 (32 on a 32-bit machine) so we get 1024 instead of the expected 1000; localbitprec bounds the relative error exactly as specified in functions that support that granularity (e.g. lfun), and rounded to the next multiple of 64 (resp. 32) everywhere else.

Warning. Changing realbitprecision or realprecision in programs is deprecated in favor of localbitprec and localprec. Think about the realprecision and realbitprecision defaults as interactive commands for the gp interpreter, best left out of GP programs. Indeed, the above rules imply that mixing both constructs yields surprising results:

```
? \p38
? localprec(19); default(realprecision,1000); Pi
%1 = 3.141592653589793239
? \p
realprecision = 1001 significant digits (1000 digits displayed)
```

Indeed, realprecision itself is ignored within localprec scope, so Pi is computed to a low accuracy. And when we leave the localprec scope, realprecision only regains precedence, it is not "restored" to the original value.

localprec()

Set the real precision to p in the dynamic scope and return p. All computations are performed as if realprecision was p: transcendental constants (e.g. Pi) and conversions from exact to floating point inexact data use p decimal digits, as well as iterative routines implicitly using a floating point accuracy as a termination criterion (e.g. solve or intnum). But realprecision itself is unaffected and is "unmasked" when we exit the dynamic (*not* lexical) scope. In effect, this is similar to

```
my(prec = default(realprecision));
default(realprecision,p);
...
default(realprecision, prec);
```

but is both less cumbersome, cleaner (no need to manipulate a global variable, which in fact never changes and is only temporarily masked) and more robust: if the above computation is interrupted or an exception occurs, realprecision will not be restored as intended.

Such localprec statements can be nested, the innermost one taking precedence as expected. Beware that localprec follows the semantic of local, not my: a subroutine called from localprec scope uses the local accuracy:

? f()=precision(1.); ? f() %2 = 38 ? localprec(19); f() %3 = 19

Warning. Changing realprecision itself in programs is now deprecated in favor of localprec. Think about the realprecision default as an interactive command for the gp interpreter, best left out of GP programs. Indeed, the above rules imply that mixing both constructs yields surprising results:

```
? \p38
? localprec(19); default(realprecision,100); Pi
%1 = 3.141592653589793239
? \p
realprecision = 115 significant digits (100 digits displayed)
```

Indeed, realprecision itself is ignored within localprec scope, so Pi is computed to a low accuracy. And when we leave localprec scope, realprecision only regains precedence, it is not "restored" to the original value.

log(precision)

Principal branch of the natural logarithm of $x \in \mathbb{C}^*$, i.e. such that $\Im(\log(x)) \in]-\pi,\pi]$. The branch cut lies along the negative real axis, continuous with quadrant 2, i.e. such that $\lim_{b\to 0^+} \log(a+bi) = \log a$ for $a \in \mathbb{R}^*$. The result is complex (with imaginary part equal to π) if $x \in \mathbb{R}$ and x < 0. In general, the algorithm uses the formula

$$\log(x) (\pi)/(2agm(1, 4/s)) - m \log 2$$
,

if $s = x2^m$ is large enough. (The result is exact to B bits provided $s > 2^{B/2}$.) At low accuracies, the series expansion near 1 is used.

p-adic arguments are also accepted for *x*, with the convention that $\log(p) = 0$. Hence in particular $\exp(\log(x))/x$ is not in general equal to 1 but to a (p-1)-th root of unity (or 1 if p = 2) times a power of *p*.

log1p(precision)

Return $\log(1+x)$, computed in a way that is also accurate when the real part of x is near 0. This is the reciprocal function of $\exp(1) = \exp(x) - 1$.

```
? default(realprecision, 10000); x = Pi*1e-100;
? (expm1(log1p(x)) - x) / x
%2 = -7.668242895059371866 E-10019
? (log1p(expm1(x)) - x) / x
%3 = -7.668242895059371866 E-10019
```

When x is small, this function is both faster and more accurate than log(1 + x):

```
? \p38
? x = 1e-20;
? localprec(100); c = log1p(x); \\ reference point
? a = log1p(x); abs((a - c)/c)
%6 = 0.E-38
? b = log(1+x); abs((b - c)/c) \\ slightly less accurate
%7 = 1.5930919111324522770 E-38
? for (i=1,10^5,log1p(x))
time = 81 ms.
? for (i=1,10^5,log(1+x))
time = 100 ms. \\ slower, too
```

logint(b, z)

Return the largest integer e so that $b^e \le x$, where the parameters b > 1 and x > 0 are both integers. If the parameter z is present, set it to b^e .

```
? logint(1000, 2)
%1 = 9
? 2^9
%2 = 512
? logint(1000, 2, &z)
%3 = 9
? z
%4 = 512
```

The number of digits used to write b in base x is 1 + logint(x,b):

```
? #digits(1000!, 10)
%5 = 2568
? logint(1000!, 10)
%6 = 2567
```

This function may conveniently replace

floor(log(x) / log(b))

which may not give the correct answer since PARI does not guarantee exact rounding.

```
mapdelete(_arg1)
```

Removes x from the domain of the map M.

```
? M = Map(["a",1; "b",3; "c",7]);
? mapdelete(M,"b");
? Mat(M)
["a" 1]
```

["c" 7]

mapget(x)

Returns the image of x by the map M.

```
? M=Map(["a",23;"b",43]);
? mapget(M,"a")
%2 = 23
? mapget(M,"b")
%3 = 43
```

Raises an exception when the key x is not present in M.

```
? mapget(M,"c")
 *** at top-level: mapget(M,"c")
 *** ^-----
 *** mapget: nonexistent component in mapget: index not in map
```

mapisdefined(x, z)

Returns true (1) if x has an image by the map M, false (0) otherwise. If z is present, set z to the image of x, if it exists.

```
? M1 = Map([1, 10; 2, 20]);
? mapisdefined(M1,3)
%1 = 0
? mapisdefined(M1, 1, &z)
%2 = 1
? z
%3 = 10
```

```
? M2 = Map(); N = 19;
? for (a=0, N-1, mapput(M2, a^3%N, a));
? {for (a=0, N-1,
    if (mapisdefined(M2, a, &b),
    printf("%d is the cube of %d mod %d\n",a,b,N)));}
0 is the cube of 0 mod 19
1 is the cube of 11 mod 19
7 is the cube of 11 mod 19
7 is the cube of 14 mod 19
11 is the cube of 17 mod 19
12 is the cube of 15 mod 19
18 is the cube of 18 mod 19
```

mapput(y, _arg2)

Associates x to y in the map M. The value y can be retrieved with mapget.

```
? M = Map();
? mapput(~M, "foo", 23);
? mapput(~M, 7718, "bill");
? mapget(M, "foo")
%4 = 23
```

```
? mapget(M, 7718)
%5 = "bill"
? Vec(M) \\ keys
%6 = [7718, "foo"]
? Mat(M)
%7 =
[ 7718 "bill"]
["foo" 23]
```

matadjoint(flag)

adjoint matrix of M, i.e. a matrix N of cofactors of M, satisfying $M * N = \det(M) * \operatorname{Id}$. M must be a (not necessarily invertible) square matrix of dimension n. If flag is 0 or omitted, we try to use Leverrier-Faddeev's algorithm, which assumes that n! invertible. If it fails or flag = 1, compute T = charpoly(M) independently first and return $(-1)^{n-1}(T(x) - T(0))/x$ evaluated at M.

? a = [1,2,3;3,4,5;6,7,8] * Mod(1,4); ? matadjoint(a) %2 = [Mod(1, 4) Mod(1, 4) Mod(2, 4)] [Mod(2, 4) Mod(2, 4) Mod(0, 4)] [Mod(1, 4) Mod(1, 4) Mod(2, 4)]

Both algorithms use $O(n^4)$ operations in the base ring. Over a field, they are usually slower than computing the characteristic polynomial or the inverse of M directly.

matalgtobasis(x)

This function is deprecated, use apply.

nf being a number field in nfinit format, and x a (row or column) vector or matrix, apply nfalgtobasis to each entry of x.

matbasistoalg(x)

This function is deprecated, use apply.

nf being a number field in nfinit format, and x a (row or column) vector or matrix, apply nfbasistoalg to each entry of x.

matcompanion()

The left companion matrix to the nonzero polynomial x.

matconcat()

Returns a t_MAT built from the entries of v, which may be a t_VEC (concatenate horizontally), a t_COL (concatenate vertically), or a t_MAT (concatenate vertically each column, and concatenate vertically the resulting matrices). The entries of v are always considered as matrices: they can themselves be t_VEC (seen as a row matrix), a t_COL seen as a column matrix), a t_MAT, or a scalar (seen as an 1x1 matrix).

```
? A=[1,2;3,4]; B=[5,6]~; C=[7,8]; D=9;
? matconcat([A, B]) \\ horizontal
%1 =
[1 2 5]
```

```
[3 4 6]
? matconcat([A, C]~) \\ vertical
%2 =
[1 2]
[3 4]
[7 8]
? matconcat([A, B; C, D]) \\ block matrix
%3 =
[1 2 5]
[3 4 6]
[7 8 9]
```

If the dimensions of the entries to concatenate do not match up, the above rules are extended as follows:

- each entry $v_{i,j}$ of v has a natural length and height: 1x1 for a scalar, 1xn for a t_VEC of length n, nx1 for a t_COL, mxn for an mxn t_MAT
- let H_i be the maximum over j of the lengths of the $v_{i,j}$, let L_j be the maximum over i of the heights of the $v_{i,j}$. The dimensions of the (i, j)-th block in the concatenated matrix are $H_i x L_j$.
- a scalar $s = v_{i,j}$ is considered as s times an identity matrix of the block dimension $\min(H_i, L_j)$
- blocks are extended by 0 columns on the right and 0 rows at the bottom, as needed.

```
? matconcat([1, [2,3]~, [4,5,6]~]) \\ horizontal
%4 =
[1 2 4]
[0 3 5]
[0 0 6]
? matconcat([1, [2,3], [4,5,6]]~) \\ vertical
%5 =
[1 \ 0 \ 0]
[2 3 0]
[4 5 6]
? matconcat([B, C; A, D]) \\ block matrix
%6 =
[5 0 7 8]
[6 0 0 0]
[1 2 9 0]
[3 4 0 9]
? U=[1,2;3,4]; V=[1,2,3;4,5,6;7,8,9];
? matconcat(matdiagonal([U, V])) \\ block diagonal
%7 =
```

[1	2	0	0	0]	
[3	4	0	0	0]	
[0]	0	1	2	3]	
[0]	0	4	5	6]	
[0]	0	7	8	9]	

matdet(flag)

Determinant of the square matrix x.

If flag = 0, uses an appropriate algorithm depending on the coefficients:

- integer entries: modular method due to Dixon, Pernet and Stein.
- real or *p*-adic entries: classical Gaussian elimination using maximal pivot.
- intmod entries: classical Gaussian elimination using first nonzero pivot.
- other cases: Gauss-Bareiss.

If flag = 1, uses classical Gaussian elimination with appropriate pivoting strategy (maximal pivot for real or *p*-adic coefficients). This is usually worse than the default.

matdetint()

Let B be an mxn matrix with integer coefficients. The *determinant* D of the lattice generated by the columns of B is the square root of det (B^TB) if B has maximal rank m, and 0 otherwise.

This function uses the Gauss-Bareiss algorithm to compute a positive *multiple* of D. When B is square, the function actually returns $D = || \det B ||$.

This function is useful in conjunction with mathnfmod, which needs to know such a multiple. If the rank is maximal but the matrix is nonsquare, you can obtain D exactly using

```
matdet( mathnfmod(B, matdetint(B)) )
```

Note that as soon as one of the dimensions gets large (m or n is larger than 20, say), it will often be much faster to use mathnf(B, 1) or mathnf(B, 4) directly.

matdetmod(d)

Given a matrix x with t_INT entries and d an arbitrary positive integer, return the determinant of x modulo d.

? A = [4,2,3; 4,5,6; 7,8,9] ? matdetmod(A,27) %2 = 9

Note that using the generic function matdet on a matrix with t_INTMOD entries uses Gaussian reduction and will fail in general when the modulus is not prime.

```
? matdet(A * Mod(1,27))
   *** at top-level: matdet(A*Mod(1,27))
   *** ^------
   *** matdet: impossible inverse in Fl_inv: Mod(3, 27).
```

matdiagonal()

x being a vector, creates the diagonal matrix whose diagonal entries are those of x.

? matdiagonal([1,2,3]); %1 = [1 0 0] [0 2 0] [0 0 3]

Block diagonal matrices are easily created using matconcat:

```
? U=[1,2;3,4]; V=[1,2,3;4,5,6;7,8,9];
? matconcat(matdiagonal([U, V]))
%1 =
[1 2 0 0 0]
[3 4 0 0 0]
[0 0 1 2 3]
[0 0 4 5 6]
[0 0 7 8 9]
```

mateigen(flag, precision)

Returns the (complex) eigenvectors of x as columns of a matrix. If flag = 1, return [L, H], where L contains the eigenvalues and H the corresponding eigenvectors; multiple eigenvalues are repeated according to the eigenspace dimension (which may be less than the eigenvalue multiplicity in the characteristic polynomial).

This function first computes the characteristic polynomial of x and approximates its complex roots (λ_i) , then tries to compute the eigenspaces as kernels of the $x - \lambda_i$. This algorithm is ill-conditioned and is likely to miss kernel vectors if some roots of the characteristic polynomial are close, in particular if it has multiple roots.

```
? A = [13,2; 10,14]; mateigen(A)
%1 =
[-1/2 2/5]
[ 1 1]
? [L,H] = mateigen(A, 1);
? L
%3 = [9, 18]
? H
%4 =
[-1/2 2/5]
[ 1 1]
? A * H == H * matdiagonal(L)
%5 = 1
```

For symmetric matrices, use qfjacobi instead; for Hermitian matrices, compute

A = real(x); B = imag(x); y = matconcat([A, -B; B, A]);

and apply qfjacobi to y.

matfrobenius(flag, v)

Returns the Frobenius form of the square matrix M. If flag = 1, returns only the elementary divisors as a vector of polynomials in the variable v. If flag = 2, returns a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.

mathess()

Returns a matrix similar to the square matrix x, which is in upper Hessenberg form (zero entries below the first subdiagonal).

mathnf(flag)

Let R be a Euclidean ring, equal to \mathbb{Z} or to K[X] for some field K. If M is a (not necessarily square) matrix with entries in R, this routine finds the *upper triangular* Hermite normal form of M. If the rank of M is equal to its number of rows, this is a square matrix. In general, the columns of the result form a basis of the R-module spanned by the columns of M.

The values of flag are:

- 0 (default): only return the Hermite normal form H
- 1 (complete output): return [H, U], where H is the Hermite normal form of M, and U is a transformation matrix such that MU = [0||H]. The matrix U belongs to GL(R). When M has a large kernel, the entries of U are in general huge.

For these two values, we use a naive algorithm, which behaves well in small dimension only. Larger values correspond to different algorithms, are restricted to *integer* matrices, and all output the unimodular matrix U. From now on all matrices have integral entries.

• flag = 4, returns [H, U] as in "complete output" above, using a variant of LLL reduction along the way. The matrix U is provably small in the L_2 sense, and often close to optimal; but the reduction is in general slow, although provably polynomial-time.

If flag = 5, uses Batut's algorithm and output [H, U, P], such that H and U are as before and P is a permutation of the rows such that P applied to MU gives H. This is in general faster than flag = 4 but the matrix U is usually worse; it is heuristically smaller than with the default algorithm.

When the matrix is dense and the dimension is large (bigger than 100, say), flag = 4 will be fastest. When M has maximal rank, then

H = mathnfmod(M, matdetint(M))

will be even faster. You can then recover U as $M^{-1}H$.

```
? M = matrix(3,4,i,j,random([-5,5]))
%1 =
[ 0 2 3 0]
[-5 3 -5 -5]
[ 4 3 -5 4]
? [H,U] = mathnf(M, 1);
```

? U %3 = [-1 0 -1 0]		
[0 5 3 2]		
[0 3 1 1]		
[1 0 0 0]		
? H %5 = [19 9 7]		
[0 9 1]		
[0 0 1]		
? M*U %6 = [0 19 9 7]		
[0 0 9 1]		
[0 0 0 1]		

For convenience, M is allowed to be a t_VEC, which is then automatically converted to a t_MAT, as per the Mat function. For instance to solve the generalized extended gcd problem, one may use

? v = [116085838, 181081878, 314252913,10346840]; ? [H,U] = mathnf(v, 1); ? U %2 = [103 -603 15 -88] [-146 13 -1208 352] [58 220 678 -167] [-362 -144 381 -101] ? v*U %3 = [0, 0, 0, 1]

This also allows to input a matrix as a t_VEC of t_COL s of the same length (which Mat would concatenate to the t_MAT having those columns):

? v = [[1,0,4]~, [3,3,4]~, [0,-4,-5]~]; mathnf(v) %1 = [47 32 12] [0 1 0] [0 0 1]

mathnfmod(d)

If x is a (not necessarily square) matrix of maximal rank with integer entries, and d is a multiple of the (nonzero) determinant of the lattice spanned by the columns of x, finds the *upper triangular* Hermite normal form of x.

If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x. Even when d is known, this is in general slower than mathnf but uses much less memory.

mathnfmodid(d)

Outputs the (upper triangular) Hermite normal form of x concatenated with the diagonal matrix with diagonal d. Assumes that x has integer entries. Variant: if d is an integer instead of a vector, concatenate d times the identity matrix.

```
? m=[0,7;-1,0;-1,-1]
%1 =
[ 0 7]
[-1 0]
[-1 -1]
? mathnfmodid(m, [6,2,2])
%2 =
[2 1 1]
[0 1 0]
[0 0 1]
? mathnfmodid(m, 10)
%3 =
[10 7 3]
[ 0 1 0]
[ 0 0 1]
```

mathouseholder(v)

applies a sequence Q of Householder transforms, as returned by matqr(M, 1) to the vector or matrix v.

```
? m = [2,1; 3,2]; \\ some random matrix
? [Q,R] = matqr(m);
? Q
%3 =
[-0.554... -0.832...]
[-0.832... 0.554...]
? R
%4 =
[-3.605... -2.218...]
[0 0.277...]
? v = [1, 2]~; \\ some random vector
? Q * v
```
%6 = [-2.218..., 0.277...]~
? [q,r] = matqr(m, 1);
? exponent(r - R) \\ r is the same as R
%8 = -128
? q \\ but q has a different structure
%9 = [[0.0494..., [5.605..., 3]]]]
? mathouseholder(q, v) \\ applied to v
%10 = [-2.218..., 0.277...]~

The point of the Householder structure is that it efficiently represents the linear operator v : -- > Qv in a more stable way than expanding the matrix Q:

```
? m = mathilbert(20); v = vectorv(20,i,i^2+1);
? [Q,R] = matqr(m);
? [q,r] = matqr(m, 1);
? \p100
? [q2,r2] = matqr(m, 1); \\ recompute at higher accuracy
? exponent(R - r)
%5 = -127
? exponent(R - r2)
%6 = -127
? exponent(mathouseholder(q,v) - mathouseholder(q2,v))
%7 = -119
? exponent(Q*v - mathouseholder(q2,v))
%8 = 9
```

We see that R is OK with or without a flag to matqr but that multiplying by Q is considerably less precise than applying the sequence of Householder transforms encoded by q.

matimage(flag)

Gives a basis for the image of the matrix x as columns of a matrix. A priori the matrix can have entries of any type. If flag = 0, use standard Gauss pivot. If flag = 1, use matsupplement (much slower: keep the default flag!).

matimagecompl()

Gives the vector of the column indices which are not extracted by the function matimage, as a permutation (t_VECSMALL). Hence the number of components of matimagecompl(x) plus the number of columns of matimage(x) is equal to the number of columns of the matrix x.

matimagemod(d, U)

Gives a Howell basis (unique representation for submodules of $(\mathbb{Z}/d\mathbb{Z})^n$) for the image of the matrix x modulo d as columns of a matrix H. The matrix x must have t_INT entries, and d can be an arbitrary positive integer. If U is present, set it to a matrix such that AU = H.

? A = [2,1;0,2]; ? matimagemod(A,6,&U) %2 = [1 0] [0 2] ? U

%3 = [5 1] [3 4] ? (A*U)%6 %4 = [1 0] [0 2]

Caveat. In general the number of columns of the Howell form is not the minimal number of generators of the submodule. Example:

? matimagemod([1;2],4)
%5 =
[2 1]
[0 2]

Caveat 2. In general the matrix U is not invertible, even if A and H have the same size. Example:

? matimagemod([4,1;0,4],8,&U)
%6 =
[2 1]
[0 4]
? U
%7 =
[0 0]
[2 1]

matindexrank()

M being a matrix of rank *r*, returns a vector with two t_VECSMALL components *y* and *z* of length *r* giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these two vectors using vecextract(M, y, z) is invertible. The vectors *y* and *z* are sorted in increasing order.

matintersect(y)

x and y being two matrices with the same number of rows each of whose columns are independent, finds a basis of the vector space equal to the intersection of the spaces spanned by the columns of x and y respectively. The faster function idealintersect can be used to intersect fractional ideals (projective \mathbb{Z}_K modules of rank 1); the slower but more general function nfhnf can be used to intersect general \mathbb{Z}_K -modules.

matinverseimage(y)

Given a matrix x and a column vector or matrix y, returns a preimage z of y by x if one exists (i.e such that xz = y), an empty vector or matrix otherwise. The complete inverse image is z + Kerx, where a basis of the kernel of x may be obtained by matker.

? M = [1,2;2,4]; ? matinverseimage(M, [1,2]~) %2 = [1, 0]~

```
? matinverseimage(M, [3,4]~)
%3 = []~ \\ no solution
? matinverseimage(M, [1,3,6;2,6,12])
%4 =
[1 3 6]
[0 0 0]
? matinverseimage(M, [1,2;3,4])
%5 = [;] \\ no solution
? K = matker(M)
%6 =
[-2]
[1]
```

matinvmod(d)

Computes a left inverse of the matrix x modulo d. The matrix x must have t_INT entries, and d can be an arbitrary positive integer.

```
? A = [3,1,2;1,2,1;3,1,1];
? U = matinvmod(A,6)
%2 =
[1 1 3]
[2 3 5]
[1 0 5]
? (U*A)%6
%3 =
[1 0 0]
[0 1 0]
[0 0 1]
? matinvmod(A,5)
*** at top-level: matinvmod(A,5)
**** at top-level: matinvmod(A,5)
**** matinvmod: impossible inverse in gen_inv: 0.
```

matisdiagonal()

Returns true (1) if x is a diagonal matrix, false (0) if not.

matker(flag)

Gives a basis for the kernel of the matrix x as columns of a matrix. The matrix can have entries of any type, provided they are compatible with the generic arithmetic operations (+, x and /).

If x is known to have integral entries, set flag = 1.

matkerint(flag)

Gives an LLL-reduced \mathbb{Z} -basis for the lattice equal to the kernel of the matrix x with rational entries. *flag* is deprecated, kept for backward compatibility.

matkermod(d, im)

Gives a Howell basis (unique representation for submodules of $(\mathbb{Z}/d\mathbb{Z})^n$, cf. matimagemod) for the kernel of the matrix x modulo d as columns of a matrix. The matrix x must have t_INT entries, and d can be an arbitrary positive integer. If im is present, set it to a basis of the image of x (which is computed on the way).

```
? A = [1,2,3;5,1,4]
%1 =
[1 2 3]
[5 1 4]
? K = matkermod(A,6)
%2 =
[2 1]
[2 1]
[0 3]
? (A*K)%6
%3 =
[0 0]
[0 0]
```

matmuldiagonal(d)

Product of the matrix x by the diagonal matrix whose diagonal entries are those of the vector d. Equivalent to, but much faster than x * matdiagonal(d).

matmultodiagonal(y)

Product of the matrices x and y assuming that the result is a diagonal matrix. Much faster than x * y in that case. The result is undefined if x * y is not diagonal.

matpermanent()

Permanent of the square matrix x using Ryser's formula in Gray code order.

```
? n = 20; m = matrix(n,n,i,j, i!=j);
? matpermanent(m)
%2 = 895014631192902121
? n! * sum(i=0,n, (-1)^i/i!)
%3 = 895014631192902121
```

This function runs in time $O(2^n n)$ for a matrix of size n and is not implemented for n large.

matqr(flag, precision)

Returns [Q, R], the QR-decomposition of the square invertible matrix M with real entries: Q is orthogonal and R upper triangular. If flag = 1, the orthogonal matrix is returned as a sequence of Householder transforms: applying such a sequence is stabler and faster than multiplication by the corresponding Q matrix. More precisely, if

```
[Q,R] = matqr(M);
[q,r] = matqr(M, 1);
```

then r = R and mathouseholder(q, M) is (close to) R; furthermore

```
mathouseholder(q, matid(#M)) == Q~
```

the inverse of Q. This function raises an error if the precision is too low or x is singular.

matrank()

Rank of the matrix x.

matreduce()

Let m be a factorization matrix, i.e., a 2-column matrix whose columns contains arbitrary "generators" and integer "exponents" respectively. Returns the canonical form of m: the first column is sorted with unique elements and the second one contains the merged "exponents" (exponents of identical entries in the first column of m are added, rows attached to 0 exponents are deleted). The generators are sorted with respect to the universal cmp routine; in particular, this function is the identity on true integer factorization matrices, but not on other factorizations (in products of polynomials or maximal ideals, say). It is idempotent.

For convenience, this function also allows a vector m, which is handled as a factorization with all exponents equal to 1, as in factorback.

? / %1 [x	A=[x,2;y,4]; = 2]	B=[x,-2;	y,3;	3,	4];	C=matconcat([A,B]~)
[у	4]					
[x	-2]					
[y	3]					
[3	4]					
?r %2 [3	<pre>matreduce(C) = 4]</pre>					
[y	7]					
?r %3 [x	matreduce([x = 4]	, x , y , x , z , z	x,y])	\\	vect	tor argument
[y	2]					
[z	1]					

matrixqz(p)

A being an mxn matrix in $M_{m,n}(\mathbb{Q})$, let $Im_{\mathbb{Q}}A$ (resp. $Im_{\mathbb{Z}}A$) the \mathbb{Q} -vector space (resp. the \mathbb{Z} -module) spanned by the columns of A. This function has varying behavior depending on the sign of p:

If $p \ge 0$, A is assumed to have maximal rank $n \le m$. The function returns a matrix $B \in M_{m,n}(\mathbb{Z})$, with $Im_{\mathbb{Q}}B = Im_{\mathbb{Q}}A$, such that the GCD of all its nxn minors is coprime to p; in particular, if p = 0 (default), this GCD is 1.

If p = -1, returns a basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{Z}}A$.

If p = -2, returns a basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{O}}A$.

Caveat. (p = -1 or -2) For efficiency reason, we do not compute the HNF of the resulting basis.

```
? minors(x) = vector(#x[,1], i, matdet(x[^i,]));
? A = [3,1/7; 5,3/7; 7,5/7]; minors(A)
%1 = [4/7, 8/7, 4/7] \setminus  determinants of all 2x2 minors
? B = matrixqz(A)
%2 =
[3 1]
[5 2]
[7 3]
? minors(%)
\%3 = [1, 2, 1] \setminus B integral with coprime minors
? matrixqz(A,-1)
%4 =
[3 1]
[5 3]
[7 5]
? matrixqz(A,-2)
%5 =
[3 1]
[5 2]
[7 3]
```

matsize()

x being a vector or matrix, returns a row vector with two components, the first being the number of rows (1 for a row vector), the second the number of columns (1 for a column vector).

matsnf(flag)

If X is a (singular or nonsingular) matrix outputs the vector of elementary divisors of X, i.e. the diagonal of the Smith normal form of X, normalized so that $d_n || d_{n-1} || ... || d_1$. X must have integer or polynomial entries; in the latter case, X must be a square matrix.

The binary digits of *flag* mean:

1 (complete output): if set, outputs [U, V, D], where U and V are two unimodular matrices such that UXV is the diagonal matrix D. Otherwise output only the diagonal of D. If X is not a square matrix, then D will be a square diagonal matrix padded with zeros on the left or the top.

4 (cleanup): if set, cleans up the output. This means that elementary divisors equal to 1 will be deleted, i.e. outputs a shortened vector D' instead of D. If complete output was required, returns [U', V', D'] so that U'XV' = D' holds. If this flag is set, X is allowed to be of the form *vector of elementary divisors' or :math: `[U,V,D]* as would normally be output with the cleanup flag unset.

matsolve(B)

Let M be a left-invertible matrix and B a column vector such that there exists a solution X to the system of linear equations MX = B; return the (unique) solution X. This has the same effect as, but is faster, than $M^{-1} * B$. Uses Dixon p-adic lifting method if M and B are integral and Gaussian elimination otherwise. When there is no solution, the function returns an X such that MX - B is nonzero although it has at least #M zero entries:

```
? M = [1,2;3,4;5,6];
? B = [4,6,8]~; X = matsolve(M, B)
%2 = [-2, 3]~
? M*X == B
%3 = 1
? B = [1,2,4]~; X = matsolve(M, [1,2,4]~)
%4 = [0, 1/2]~
? M*X - B
%5 = [0, 0, -1]~
```

Raises an exception if M is not left-invertible, even if there is a solution:

```
? M = [1,1;1,1]; matsolve(M, [1,1]~)
   *** at top-level: matsolve(M,[1,1]~)
   *** ^_____
*** matsolve: impossible inverse in gauss: [1, 1; 1, 1].
```

The function also works when B is a matrix and we return the unique matrix solution X provided it exists.

matsolvemod(D, B, flag)

M being any integral matrix, *D* a column vector of nonnegative integer moduli, and *B* an integral column vector, gives an integer solution to the system of congruences $\sum_{i} m_{i,j} x_j = b_i (modd_i)$ if one exists, otherwise returns zero. Shorthand notation: *B* (resp. *D*) can be given as a single integer, in which case all the b_i (resp. d_i) above are taken to be equal to *B* (resp. *D*).

```
? M = [1,2;3,4];
? matsolvemod(M, [3,4]~, [1,2]~)
%2 = [10, 0]~
? matsolvemod(M, 3, 1) \\ M X = [1,1]~ over F_3
%3 = [2, 1]~
? matsolvemod(M, [3,0]~, [1,2]~) \\ x + 2y = 1 (mod 3), 3x + 4y = 2 (in Z)
%4 = [6, -4]~
```

If flag = 1, all solutions are returned in the form of a two-component row vector [x, u], where x is an integer solution to the system of congruences and u is a matrix whose columns give a basis of the homogeneous system (so that all solutions can be obtained by adding x to any linear combination of columns of u). If no solution exists, returns zero.

matsupplement()

Assuming that the columns of the matrix x are linearly independent (if they are not, an error message is issued), finds a square invertible matrix whose first columns are the columns of x, i.e. supplement the columns of x to a basis of the whole space.

? matsupplement([1;2])
%1 =
[1 0]
[2 1]

Raises an error if x has 0 columns, since (due to a long standing design bug), the dimension of the ambient space (the number of rows) is unknown in this case:

```
? matsupplement(matrix(2,0))
    *** at top-level: matsupplement(matrix
```

(continued from previous page)

```
*** matsupplement: sorry, suppl [empty matrix] is not yet implemented.
```

mattranspose()

Transpose of x (also x). This has an effect only on vectors and matrices.

max(y)

Creates the maximum of x and y when they can be compared.

mfEH()

k being in $1/2 + \mathbb{Z}_{>=0}$, return the mf structure corresponding to the Cohen-Eisenstein series H_k of weight k on $\Gamma_0(4)$.

```
? H = mfEH(13/2); mfcoefs(H,4)
%1 = [691/32760, -1/252, 0, 0, -2017/252]
```

The coefficients of H are given by the Cohen-Hurwitz function H(k-1/2, N) and can be obtained for moderately large values of N (the algorithm uses O(N) time):

```
? mfcoef(H,10^5+1)
time = 55 ms.
%2 = -12514802881532791504208348
? mfcoef(H,10^7+1)
time = 6,044 ms.
%3 = -1251433416009877455212672599325104476
```

mfTheta()

The unary theta function corresponding to the primitive Dirichlet character ψ . Its level is $4F(\psi)^2$ and its weight is $1 - \psi(-1)/2$.

```
? Ser(mfcoefs(mfTheta(),30))
%1 = 1 + 2*x + 2*x^4 + 2*x^9 + 2*x^16 + 2*x^25 + 0(x^31)
? f = mfTheta(8); Ser(mfcoefs(f,30))
%2 = 2*x - 2*x^9 - 2*x^25 + 0(x^31)
? mfparams(f)
%3 = [256, 1/2, 8, y, t + 1]
? g = mfTheta(-8); Ser(mfcoefs(g,30))
%4 = 2*x + 6*x^9 - 10*x^25 + 0(x^31)
? mfparams(g)
%5 = [256, 3/2, 8, y, t + 1]
? h = mfTheta(Mod(2,5)); mfparams(h)
%6 = [100, 3/2, Mod(7, 20), y, t^2 + 1]
```

mfatkin(f)

Given a mfatk output by mfatk = mfatkininit(mf,Q) and a modular form f belonging to the pace mf, returns the modular form $g = Cxf || W_Q$, where C = mfatk[3] is a normalizing constant such that g has the same field of coefficients as f; mfatk[3] gives the constant C, and mfatk[1] gives the modular form space to which gbelongs (or is set to 0 if it is mf).

```
? mf = mfinit([35,2],0); [f] = mfbasis(mf);
? mfcoefs(f, 4)
%2 = [0, 3, -1, 0, 3]
? mfatk = mfatkininit(mf,7);
? g = mfatkin(mfatk, f); mfcoefs(g, 4)
%4 = [0, 1, -1, -2, 7]
? mfatk = mfatkininit(mf,35);
? g = mfatkin(mfatk, f); mfcoefs(g, 4)
%6 = [0, -3, 1, 0, -3]
```

mfatkineigenvalues(Q, precision)

Given a modular form space mf of integral weight k and a primitive divisor Q of the level N of mf, outputs the Atkin-Lehner eigenvalues of w_Q on the new space, grouped by orbit. If the Nebentypus χ of mf is a (trivial or) quadratic character defined modulo N/Q, the result is rounded and the eigenvalues are i^k .

```
? mf = mfinit([35,2],0); mffields(mf)
%1 = [y, y^2 - y - 4] \\ two orbits, dimension 1 and 2
? mfatkineigenvalues(mf,5)
%2 = [[1], [-1, -1]]
? mf = mfinit([12,7,Mod(3,4)],0);
? mfatkineigenvalues(mf,3)
%4 = [[I, -I, -I, I, I, -I]] \\ one orbit
```

To obtain the eigenvalues on a larger space than the new space, e.g., the full space, you can directly call [mfB, M, C] = mfatkininit and compute the eigenvalues as the roots of the characteristic polynomial of M/C, by dividing the roots of charpoly(M) by C. Note that the characteristic polynomial is computed exactly since M has coefficients in $\mathbb{Q}(\chi)$, whereas C may be given by a complex number. If the coefficients of the characteristic polynomial are polmods modulo T they must be embedded to \mathbb{C} first using subst(lift(), t, exp(2*I*Pi/n)), when T is poliscyclo(n); note that T = mf.mod.

mfatkininit(Q, precision)

Given a modular form space with parameters N, k, χ and a primitive divisor Q of the level N, initializes data necessary for working with the Atkin-Lehner operator W_Q , for now only the function mfatkin. We write $\chi \chi_Q \chi_{N/Q}$ where the two characters are primitive with (coprime) conductors dividing Q and N/Q respectively. For $F \in M_k(\Gamma_0(N), \chi)$, the form $F || W_Q$ still has level N and weight k but its Nebentypus may no longer be χ : it becomes $\overline{\chi_Q} \chi_{N/Q}$) if k is integral and $\overline{\chi_Q} \chi_{N/Q})(4Q/.)$ if not.

The result is a technical 4-component vector [mfB, MC, C, mf], where

- mfB encodes the modular form space to which $F || W_Q$ belongs when $F \in M_k(\Gamma_0(N), \chi)$: an mfinit corresponding to a new Nebentypus or the integer 0 when the character does not change. This does not depend on F.
- MC is the matrix of W_Q on the bases of mf and mfB multiplied by a normalizing constant C(k, χ, Q). This matrix has polmod coefficients in Q(χ).
- C is the complex constant $C(k, \chi, Q)$. For k integral, let $A(k, \chi, Q) = Q^{\varepsilon}/g(\chi_Q)$, where $\varepsilon = 0$ for k even and 1/2 for k odd and where $g(\chi_Q)$ is the Gauss sum attached to χ_Q). (A similar, more complicated, definition holds in half-integral weight depending on the parity of k 1/2.) Then if M denotes the matrix of W_Q on the bases of mf and mfB, A.M has coefficients in $\mathbb{Q}(\chi)$. If A is rational, we let C = 1 and C = A as a floating point complex number otherwise, and finally MC := M.C.

```
? mf=mfinit([32,4],0); [mfB,MC,C]=mfatkininit(mf,32); MC
%1 =
[5/16 11/2 55/8]
```

```
[ 1/8 0 -5/4]
[1/32 -1/4 11/16]
? C
%2 = 1
? mf=mfinit([32,4,8],0); [mfB,MC,C]=mfatkininit(mf,32); MC
%3 =
[ 1/8 -7/4]
[-1/16 -1/8]
? C
%4 = 0.35355339059327376220042218105242451964
? algdep(C,2) \\ C = 1/sqrt(8)
%5 = 8*x^2 - 1
```

mfbasis(space)

If NK = [N, k, CHI] as in mfinit, gives a basis of the corresponding subspace of $M_k(\Gamma_0(N), \chi)$. NK can also be the output of mfinit, in which case space can be omitted. To obtain the eigenforms, use mfeigenbasis.

If space is a full space M_k , the output is the union of first, a basis of the space of Eisenstein series, and second, a basis of the cuspidal space.

```
? see(L) = apply(f->mfcoefs(f,3), L);
? mf = mfinit([35,2],0);
? see( mfbasis(mf) )
%2 = [[0, 3, -1, 0], [0, -1, 9, -8], [0, 0, -8, 10]]
? see( mfeigenbasis(mf) )
%3 = [[0, 1, 0, 1], [Mod(0, z^2 - z - 4), Mod(1, z^2 - z - 4), \
Mod(-z, z^2 - z - 4), Mod(z - 1, z^2 - z - 4)]]
? mf = mfinit([35,2]);
? see( mfbasis(mf) )
%5 = [[1/6, 1, 3, 4], [1/4, 1, 3, 4], [17/12, 1, 3, 4], \
[0, 3, -1, 0], [0, -1, 9, -8], [0, 0, -8, 10]]
? see( mfbasis([48,4],0) )
%6 = [[0, 3, 0, -3], [0, -3, 0, 27], [0, 2, 0, 30]]
```

mfbd(d)

F being a generalized modular form, return B(d)(F), where B(d) is the expanding operator $\tau : -- > d\tau$.

? D2=mfbd(mfDelta(),2); mfcoefs(D2, 6) %1 = [0, 0, 1, 0, -24, 0, 252]

mfbracket(G, m)

Compute the m-th Rankin-Cohen bracket of the generalized modular forms F and G.

? E4 = mfEk(4); E6 = mfEk(6); ? D1 = mfbracket(E4,E4,2); mfcoefs(D1,5)/4800 %2 = [0, 1, -24, 252, -1472, 4830] ? D2 = mfbracket(E4,E6,1); mfcoefs(D2,10)/(-3456) %3 = [0, 1, -24, 252, -1472, 4830]

mfcoef(n)

Compute the *n*-th Fourier coefficient a(n) of the generalized modular form *F*. Note that this is the n + 1-st component of the vector mfcoefs(F,n) as well as the second component of mfcoefs(F,1,n).

```
? mfcoef(mfDelta(),10)
%1 = -115920
```

mfcoefs(n, d)

Compute the vector of Fourier coefficients [a[0], a[d], ..., a[nd]] of the generalized modular form F; d must be positive and d = 1 by default.

```
? D = mfDelta();
? mfcoefs(D,10)
%2 = [0, 1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
? mfcoefs(D,5,2)
%3 = [0, -24, -1472, -6048, 84480, -115920]
? mfcoef(D,10)
%4 = -115920
```

This function also applies when F is a modular form space as output by mfinit; it then returns the matrix whose columns give the Fourier expansions of the elements of mfbasis(F):

```
? mf = mfinit([1,12]);
? mfcoefs(mf,5)
%2 =
[691/65520 0]
[ 1 1]
[ 2049 -24]
[ 177148 252]
[ 4196353 -1472]
[ 48828126 4830]
```

mfconductor(F)

mf being output by mfinit for the cuspidal space and F a modular form, gives the smallest level at which F is defined. In particular, if F is cuspidal and we write $F = \sum_j B(d_j)f_j$ for new forms f_j of level N_j (see mftonew), then its conductor is the least common multiple of the $d_j N_j$.

```
? mf=mfinit([96,6],1); vF = mfbasis(mf); mfdim(mf)
%1 = 72
? vector(10,i, mfconductor(mf, vF[i]))
%2 = [3, 6, 12, 24, 48, 96, 4, 8, 12, 16]
```

mfcosets()

Let N be a positive integer. Return the list of right cosets of $\Gamma_0(N)$ Gamma, i.e., matrices $\gamma_j \in \Gamma$ such that $\Gamma = \bigsqcup_i \Gamma_0(N)\gamma_j$. The γ_j are chosen in the form [a, b; c, d] with c || N.

```
? mfcosets(4)
%1 = [[0, -1; 1, 0], [1, 0; 1, 1], [0, -1; 1, 2], [0, -1; 1, 3],\
[1, 0; 2, 1], [1, 0; 4, 1]]
```

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

```
? M = mfinit([4, 12, 1], 0); mfcosets(M)
%2 = [[0, -1; 1, 0], [1, 0; 1, 1], [0, -1; 1, 2], [0, -1; 1, 3],\
[1, 0; 2, 1], [1, 0; 4, 1]]
```

Warning. In the present implementation, the trivial coset is represented by [1, 0; N, 1] and is the last in the list.

mfcuspisregular(cusp)

In the space defined by NK = [N, k, CHI] or NK = mf, determine if cusp in canonical format (oo or denominator dividing N) is regular or not.

```
? mfcuspisregular([4,3,-4],1/2)
%1 = 0
```

mfcusps()

Let N be a positive integer. Return the list of cusps of $\Gamma_0(N)$ in the form a/b with b||N.

? mfcusps(24) %1 = [0, 1/2, 1/3, 1/4, 1/6, 1/8, 1/12, 1/24]

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

? M = mfinit([4, 12, 1], 0); mfcusps(M) %2 = [0, 1/2, 1/4]

mfcuspval(F, cusp, precision)

Valuation of modular form F in the space mf at cusp, which can be either *oo* or any rational number. The result is either a rational number or *oo* if F is zero. Let χ be the Nebentypus of the space mf; if $\mathbb{Q}(F)! = \mathbb{Q}(\chi)$, return the vector of valuations attached to the $[\mathbb{Q}(F): \mathbb{Q}(chi)]$ complex embeddings of F.

```
? T=mfTheta(); mf=mfinit([12,1/2]); mfcusps(12)
%1 = [0, 1/2, 1/3, 1/4, 1/6, 1/12]
? apply(x->mfcuspval(mf,T,x), %1)
\%2 = [0, 1/4, 0, 0, 1/4, 0]
? mf=mfinit([12,6,12],1); F=mfbasis(mf)[5];
? apply(x->mfcuspval(mf,F,x),%1)
\%4 = [1/12, 1/6, 1/2, 2/3, 1/2, 2]
? mf=mfinit([12,3,-4],1); F=mfbasis(mf)[1];
? apply(x->mfcuspval(mf,F,x),%1)
\%6 = [1/12, 1/6, 1/4, 2/3, 1/2, 1]
? mf = mfinit([625,2],0); [F] = mfeigenbasis(mf); mfparams(F)
\%7 = [625, 2, 1, y^2 - y - 1, t - 1] \setminus [Q(F):Q(chi)] = 2
? mfcuspval(mf, F, 1/25)
\%8 = [1, 2] \setminus one conjugate has valuation 1, and the other is 2
? mfcuspval(mf, F, 1/5)
\%9 = [1/25, 1/25]
```

mfcuspwidth(cusp)

Width of cusp in $\Gamma_0(N)$.

? mfcusps(12) %1 = [0, 1/2, 1/3, 1/4, 1/6, 1/12]

```
? [mfcuspwidth(12,c) | c <- mfcusps(12)]
%2 = [12, 3, 4, 3, 1, 1]
? mfcuspwidth(12, oo)
%3 = 1</pre>
```

We also allow the argument N to be a modular form space, in which case it is replaced by the level of the space:

? M = mfinit([4, 12, 1], 0); mfcuspwidth(M, 1/2) %4 = 1

mfderiv(m)

m-th formal derivative of the power series corresponding to the generalized modular form *F*, with respect to the differential operator qd/dq (default m = 1).

? D=mfDelta(); ? mfcoefs(D, 4) %2 = [0, 1, -24, 252, -1472] ? mfcoefs(mfderiv(D), 4) %3 = [0, 1, -48, 756, -5888]

mfderivE2(m)

Compute the Serre derivative $(q.d/dq)F - kE_2F/12$ of the generalized modular form F, which has weight k+2; if F is a true modular form, then its Serre derivative is also modular. If m > 1, compute the m-th iterate, of weight k + 2m.

```
? mfcoefs(mfderivE2(mfEk(4)),5)*(-3)
%1 = [1, -504, -16632, -122976, -532728]
? mfcoefs(mfEk(6),5)
%2 = [1, -504, -16632, -122976, -532728]
```

mfdescribe(G)

Gives a human-readable description of F, which is either a modular form space or a generalized modular form. If the address of G is given, puts into G the vector of parameters of the outermost operator defining F; this vector is empty if F is a leaf (an atomic object such as mfDelta(), not defined in terms of other forms) or a modular form space.

```
? E1 = mfeisenstein(4,-3,-4); mfdescribe(E1)
\%1 = "F_4(-3, -4)"
? E2 = mfeisenstein(3,5,-7); mfdescribe(E2)
\%2 = "F_3(5, -7)"
? E3 = mfderivE2(mfmul(E1,E2), 3); mfdescribe(E3,&G)
\%3 = "DERE2^3(MUL(F_4(-3, -4), F_3(5, -7)))"
? mfdescribe(G[1][1])
%4 = "MUL(F_4(-3, -4), F_3(5, -7))"
? G[2]
\%5 = 3
? for (i = 0, 4, mf = mfinit([37,4],i); print(mfdescribe(mf)));
S_4^new(G_0(37, 1))
S_4(G_0(37, 1))
S_4^old(G_0(37, 1))
E_4(G_0(37, 1))
M_4(G_0(37, 1))
```

mfdim(space)

If NK = [N, k, CHI] as in mfinit, gives the dimension of the corresponding subspace of $M_k(\Gamma_0(N), \chi)$. NK can also be the output of mfinit, in which case space must be omitted.

The subspace is described by the small integer space: 0 for the newspace $S_k^{new}(\Gamma_0(N), \chi)$, 1 for the cuspidal space S_k , 2 for the oldspace S_k^{old} , 3 for the space of Eisenstein series E_k and 4 for the full space M_k .

Wildcards. As in mfinit, *CHI* may be the wildcard 0 (all Galois orbits of characters); in this case, the output is a vector of [*order*, *conrey*, *dim*, *dimdih*] corresponding to the nontrivial spaces, where

- order is the order of the character,
- conrey its Conrey label from which the character may be recovered via znchar(conrey),
- dim the dimension of the corresponding space,
- *dimdih* the dimension of the subspace of dihedral forms corresponding to Hecke characters if k = 1 (this is not implemented for the old space and set to -1 for the time being) and 0 otherwise.

The spaces are sorted by increasing order of the character; the characters are taken up to Galois conjugation and the Conrey number is the minimal one among Galois conjugates. In weight 1, this is only implemented when the space is 0 (newspace), 1 (cusp space), 2(old space) or 3(Eisenstein series).

Wildcards for sets of characters. CHI may be a set of characters, and we return the set of [dim, dimdih].

Wildcard for :math: $M_k(Gamma_1(N))$. Additionally, the wildcard CHI = -1 is available in which case we output the total dimension of the corresponding subspace of $M_k(\Gamma_1(N))$. In weight 1, this is not implemented when the space is 4 (fullspace).

```
? mfdim([23,2], 0) \\ new space
%1 = 2
? mfdim([96,6], 0)
%2 = 10
? mfdim([10^9,4], 3) \\ Eisenstein space
%1 = 40000
? mfdim([10^9+7,4], 3)
%2 = 2
? mfdim([68,1,-1],0)
%3 = 3
? mfdim([68,1,0],0)
%4 = [[2, Mod(67, 68), 1, 1], [4, Mod(47, 68), 1, 1]]
? mfdim([124,1,0],0)
%5 = [[6, Mod(67, 124), 2, 0]]
```

This last example shows that there exists a nondihedral form of weight 1 in level 124.

mfdiv(G)

Given two generalized modular forms F and G, compute F/G assuming that the quotient will not have poles at infinity. If this is the case, use mfshift before doing the division.

```
? D = mfDelta(); \\ Delta
? H = mfpow(mfEk(4), 3);
? J = mfdiv(H, D)
*** at top-level: J=mfdiv(H,mfdeltac
*** ^------
*** mfdiv: domain error in mfdiv: ord(G) > ord(F)
? J = mfdiv(H, mfshift(D,1));
? mfcoefs(J, 4)
%4 = [1, 744, 196884, 21493760, 864299970]
```

mfeigenbasis()

Vector of the eigenforms for the space mf. The initial basis of forms computed by mfinit before splitting is also available via mfbasis.

```
? mf = mfinit([26,2],0);
? see(L) = for(i=1,#L,print(mfcoefs(L[i],6)));
? see( mfeigenbasis(mf) )
[0, 1, -1, 1, 1, -3, -1]
[0, 1, 1, -3, 1, -1, -3]
? see( mfbasis(mf) )
[0, 2, 0, -2, 2, -4, -4]
[0, -2, -4, 10, -2, 0, 8]
```

The eigenforms are internally expressed as (algebraic) linear combinations of mfbasis(mf) and it is very inefficient to compute many coefficients of those forms individually: you should rather use mfcoefs(mf) to expand the basis once and for all, then multiply by mftobasis(mf, f) for the forms you're interested in:

```
? mf = mfinit([96,6],0); B = mfeigenbasis(mf); #B
%1 = 8;
? vector(#B, i, mfcoefs(B[i],1000)); \\ expanded individually: slow
time = 7,881 ms.
? M = mfcoefs(mf, 1000); \\ initialize once
time = 982 ms.
? vector(#B, i, M * mftobasis(mf,B[i])); \\ then expand: much faster
time = 623 ms.
```

When the eigenforms are defined over an extension field of $\mathbb{Q}(\chi)$ for a nonrational character, their coefficients are hard to read and you may want to lift them or to express them in an absolute number field. In the construction below T defines $\mathbb{Q}(f)$ over \mathbb{Q} , a is the image of the generator $Mod(t, t^2 + t + 1)$ of $\mathbb{Q}(\chi)$ in $\mathbb{Q}(f)$ and y - ka is the image of the root y of f.mod:

```
? mf = mfinit([31, 2, Mod(25,31)], 0); [f] = mfeigenbasis(mf);
? f.mod
%2 = Mod(1, t<sup>2</sup> + t + 1)*y<sup>2</sup> + Mod(2*t + 2, t<sup>2</sup> + t + 1)
? v = liftpol(mfcoefs(f,5))
%3 = [0, 1, (-t - 1)*y - 1, t*y + (t + 1), (2*t + 2)*y + 1, t]
? [T,a,k] = rnfequation(mf.mod, f.mod, 1)
%4 = [y<sup>4</sup> + 2*y<sup>2</sup> + 4, Mod(-1/2*y<sup>2</sup> - 1, y<sup>4</sup> + 2*y<sup>2</sup> + 4), 0]
? liftpol(substvec(v, [t,y], [a, y-k*a]))
%5 = [0, 1, 1/2*y<sup>3</sup> - 1, -1/2*y<sup>3</sup> - 1/2*y<sup>2</sup> - y, -y<sup>3</sup> + 1, -1/2*y<sup>2</sup> - 1]
```

Beware that the meaning of y has changed in the last line is different: it now represents of root of T, no longer of f.mod (the notions coincide if k = 0 as here but it will not always be the case). This can be avoided with an extra variable substitution, for instance

```
? [T,a,k] = rnfequation(mf.mod, subst(f.mod,'y,'x), 1)
%6 = [x^4 + 2*x^2 + 4, Mod(-1/2*x^2 - 1, x^4 + 2*x^2 + 4), 0]
? liftpol(substvec(v, [t,y], [a, x-k*a]))
%7 = [0, 1, 1/2*x^3 - 1, -1/2*x^3 - 1/2*x^2 - x, -x^3 + 1, -1/2*x^2 - 1]
```

mfeigensearch(AP)

Search for a normalized rational eigen cuspform with quadratic character given restrictions on a few initial coefficients. The meaning of the parameters is as follows:

• NK governs the limits of the search: it is of the form [N, k]: search for given level N, weight k and quadratic

character; note that the character (D/.) is uniquely determined by (N, k). The level N can be replaced by a vector of allowed levels.

• AP is the search criterion, which can be omitted: a list of pairs $[..., [p, a_p], ...]$, where p is a prime number and a_p is either a t_INT (the p-th Fourier coefficient must match a_p exactly) or a t_INTMOD Mod(a, b) (the p-th coefficient must be congruent to a modulo b).

The result is a vector of newforms f matching the search criteria, sorted by increasing level then increasing ||D||.

? #mfeigensearch([[1..80],2], [[2,2],[3,-1]]) %1 = 1 ? #mfeigensearch([[1..80],2], [[2,2],[5,2]]) %2 = 1 ? v = mfeigensearch([[1..20],2], [[3,Mod(2,3)],[7,Mod(5,7)]]); #v %3 = 1 ? F=v[1]; [mfparams(F)[1], mfcoefs(F,15)] %4 = [11, [0, 1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1]]

mfembed(v, precision)

Let f be a generalized modular form with parameters $[N, k, \chi, P]$ (see mfparams, we denote $\mathbb{Q}(\chi)$ the subfield of \mathbb{C} generated by the values of χ and $\mathbb{Q}(f)$ the field of definition of f. In this context $\mathbb{Q}(\chi)$ has a single canonical complex embeding given by $s : Mod(t, polcyclo(n, t)) : - - - > \exp(2i\pi/n)$ and the number field $\mathbb{Q}(f)$ has $[\mathbb{Q}(f) : \mathbb{Q}(\chi)]$ induced embeddings attached to the complex roots of the polynomial s(P). If $\mathbb{Q}(f)$ is strictly larger than $\mathbb{Q}(\chi)$ we only allow an f which is an eigenform, produced by mfeigenbasis.

This function is meant to create embeddings of $\mathbb{Q}(f)$ and/or apply them to the object v, typically a vector of Fourier coefficients of f from mfcoefs.

- If v is omitted and f is a modular form as above, we return the embedding of Q(χ) if Q(χ) = Q(f) and a vector containing [Q(f) : Q(χ)] embeddings of Q(f) otherwise.
- If v is given, it must be a scalar in Q(f), or a vector/matrix of such, we apply the embeddings coefficientwise and return either a single result if Q(f) = Q(χ) and a vector of [Q(f) : Q(χ)] results otherwise.
- Finally f can be replaced by a single embedding produced by mfembed(f) (v was omitted) and we apply that particular embedding to v.

```
? mf = mfinit([35,2,Mod(11,35)], 0);
? [f] = mfbasis(mf);
? f.mod \setminus Q (chi) = Q (zeta_3)
\%3 = t^2 + t + 1
? v = mfcoefs(f,5); lift(v) \\ coefficients in Q (chi)
\%4 = [0, 2, -2*t - 2, 2*t, 2*t, -2*t - 2]
? mfembed(f, v) \setminus single embedding
%5 = [0, 2, -1 - 1.7320...*I, -1 + 1.73205...*I, -1 + 1.7320...*I, ...]
? [F] = mfeigenbasis(mf);
? mffields(mf)
\%7 = [y^2 + Mod(-2^*t, t^2 + t + 1)] \setminus [Q (f):Q (chi)] = 2
? V = liftpol( mfcoefs(F,5) );
\%8 = [0, 1, y + (-t - 1), (t + 1)*y + t, (-2*t - 2)*y + t, -t - 1]
? vall = mfembed(F, V); #vall
\%9 = 2 \setminus 2 embeddings, both applied to V
? vall[1] \setminus the first
%10 = [0, 1, -1.2071... - 2.0907...*I, 0.2071... - 0.3587...*I, ...]
? vall[2] \setminus and the second one
```

```
%11 = [0, 1, 0.2071... + 0.3587...*I, -1.2071... + 2.0907...*I, ...]
? vE = mfembed(F); #vE \\ same 2 embeddings
%12 = 2
? mfembed(vE[1], V) \\ apply first embedding to V
%13 = [0, 1, -1.2071... - 2.0907...*I, 0.2071... - 0.3587...*I, ...]
```

For convenience, we also allow a modular form space from mfinit instead of f, corresponding to the single embedding of $\mathbb{Q}(\chi)$.

```
? [mfB,MC,C] = mfatkininit(mf,7); MC \\ coefs in Q (chi)
%13 =
[ Mod(2/7*t, t^2 + t + 1) Mod(-1/7*t - 2/7, t^2 + t + 1)]
[Mod(-1/7*t - 2/7, t^2 + t + 1) Mod(2/7*t, t^2 + t + 1)]
? C \\ normalizing constant
%14 = 0.33863... - 0.16787*I
? M = mfembed(mf, MC) / C \\ the true matrix for the action of w_7
[-0.6294... + 0.4186...*I -0.3625... - 0.5450...*I]
[-0.3625... - 0.5450...*I -0.6294... + 0.4186...*I]
? exponent(M*conj(M) - 1) \\ M * conj(M) is close to 1
%16 = -126
```

mfeval(*F*, *vtau*, *precision*)

Computes the numerical value of the modular form F, belonging to mf, at the complex number vtau or the vector vtau of complex numbers in the completed upper-half plane. The result is given with absolute error less than 2^{-B} , where B = realbit precision.

If the field of definition $\mathbb{Q}(F)$ is larger than $\mathbb{Q}(\chi)$ then F may be embedded into \mathbb{C} in $d = [\mathbb{Q}(F) : \mathbb{Q}(\chi)]$ ways, in which case a vector of the d results is returned.

```
? mf = mfinit([11,2],0); F = mfbasis(mf)[1]; mfparams(F)
%1 = [11, 2, 1, y, t-1] \\ Q(F) = Q(chi) = Q
? mfeval(mf,F,I/2)
%2 = 0.039405471130100890402470386372028382117
? mf = mfinit([35,2],0); F = mfeigenbasis(mf)[2]; mfparams(F)
%3 = [35, 2, 1, y^2 - y - 4, t - 1] \\ [Q(F) : Q(chi)] = 2
? mfeval(mf,F,I/2)
%4 = [0.045..., 0.0385...] \\ sigma_1(F) and sigma_2(F) at I/2
? mf = mfinit([12,4],1); F = mfbasis(mf)[1];
? mfeval(mf, F, 0.318+10^(-7)*I)
%6 = 3.379... E-21 + 6.531... E-21*I \\ instantaneous !
```

In order to maximize the imaginary part of the argument, the function computes $(f||_k\gamma)(\gamma^{-1}.\tau)$ for a suitable γ not necessarily in $\Gamma_0(N)$ (in which case $f||\gamma$ is evaluated using mfslashexpansion).

```
? T = mfTheta(); mf = mfinit(T); mfeval(mf,T,[0,1/2,1,oo])
%1 = [1/2 - 1/2*I, 0, 1/2 - 1/2*I, 1]
```

mffields()

Given mf as output by mfinit with parameters (N, k, χ) , returns the vector of polynomials defining each Galois orbit of newforms over $\mathbb{Q}(\chi)$.

```
? mf = mfinit([35,2],0); mffields(mf)
%1 = [y, y^2 - y - 4]
```

Here the character is trivial so $\mathbb{Q}(\chi) = \mathbb{Q}$) and there are 3 newforms: one is rational (corresponding to y), the other two are conjugate and defined over the quadratic field $\mathbb{Q}[y]/(y^2 - y - 4)$.

```
? [G,chi] = znchar(Mod(3,35));
? zncharconductor(G,chi)
%2 = 35
? charorder(G,chi)
%3 = 12
? mf = mfinit([35, 2, [G,chi]],0); mffields(mf)
%4 = [y, y]
```

Here the character is primitive of order 12 and the two newforms are defined over $\mathbb{Q}(\chi) = \mathbb{Q}(\zeta_{12})$.

? mf = mfinit([35, 2, Mod(13,35)],0); mffields(mf) %3 = [y^2 + Mod(5*t, t^2 + 1)]

This time the character has order 4 and there are two conjugate newforms over $\mathbb{Q}(\chi) = Q(i)$.

mffromell()

E being an elliptic curve defined over Q given by an integral model in ellipsit format, computes a 3-component vector [mf,F,v], where *F* is the newform corresponding to *E* by modularity, mf is the newspace to which *F* belongs, and v gives the coefficients of *F* on mfbasis(mf).

```
? E = ellinit("26a1");
? [mf,F,co] = mffromell(E);
? co
%2 = [3/4, 1/4]~
? mfcoefs(F, 5)
%3 = [0, 1, -1, 1, 1, -3]
? ellan(E, 5)
%4 = [1, -1, 1, 1, -3]
```

mffrometaquo(flag)

Modular form corresponding to the eta quotient matrix eta. If the valuation v at infinity is fractional, return 0. If the eta quotient is not holomorphic but simply meromorphic, return 0 if flag = 0; return the eta quotient (divided by q to the power -v if v < 0, i.e., with valuation 0) if flag is set.

```
? mffrometaquo(Mat([1,1]),1)
%1 = 0
? mfcoefs(mffrometaquo(Mat([1,24])),6)
%2 = [0, 1, -24, 252, -1472, 4830, -6048]
? mfcoefs(mffrometaquo([1,1;23,1]),10)
%3 = [0, 1, -1, -1, 0, 0, 1, 0, 1, 0, 0]
? F = mffrometaquo([1,2;2,-1]); mfparams(F)
%4 = [16, 1/2, 1, y, t - 1]
? mfcoefs(F,10)
%5 = [1, -2, 0, 0, 2, 0, 0, 0, 0, -2, 0]
? mffrometaquo(Mat([1,-24]))
```

```
%6 = 0
? f = mffrometaquo(Mat([1,-24]),1); mfcoefs(f,6)
%7 = [1, 24, 324, 3200, 25650, 176256, 1073720]
```

For convenience, a t_VEC is also accepted instead of a factorization matrix with a single row:

```
? f = mffrometaquo([1,24]); \\ also valid
```

mffromlfun(precision)

Let *L* being an *L*-function in any of the lfun formats representing a self-dual modular form (for instance an eigenform). Return [NK, space, v] when mf = mfinit(NK, space) is the modular form space containing the form and mftobasis(mf, v) will represent it on the space basis. If *L* has rational coefficients, this will be enough to recognize the modular form in *mf*:

If L has inexact complex coefficients, one can for instance compute an eigenbasis for mf and check whether one of the attached L-function is reasonably close to L. In the example, we cheat by producing the L function from an eigenform in a known space, but the function does not use this information:

```
? mf = mfinit([32,6,Mod(5,32)],0);
? [poldegree(K) | K<-mffields(mf)]
%2 = [19] \\ one orbit, [Q(F) : Q(chi)] = 19
? L = lfunmf(mf)[1][1]; \\ one of the 19 L-functions attached to F
? lfunan(L,3)
%4 = [1, 5.654... - 0.1812...*I, -7.876... - 19.02...*I]
? [NK,space,v] = mffromlfun(L); NK
%5 = [32, 6, Mod(5, 32)]
? vL = concat(lfunmf(mf)); \\ L functions for all cuspidal eigenforms
? an = lfunan(L,10);
? for (i = 1, #vL, if (normlp(lfunan(vL[i],10) - an, oo) < 1e-10, print(i)));
1
```

mffromqf(P)

Q being an even integral positive definite quadratic form and P a homogeneous spherical polynomial for Q, computes a 3-component vector [mf, F, v], where F is the theta function corresponding to (Q, P), mf is the corresponding space of modular forms (from mfinit), and v gives the coefficients of F on mfbasis(mf).

```
? [mf,F,v] = mffromqf(2*matid(10)); v
%1 = [64/5, 4/5, 32/5]~
? mfcoefs(F, 5)
%2 = [1, 20, 180, 960, 3380, 8424]
? mfcoef(F, 10000) \\ number of ways of writing 10000 as sum of 10 squares
%3 = 128205250571893636
```

```
? mfcoefs(F, 10000); \\ fast !
time = 220ms
? [mf,F,v] = mffromqf([2,0;0,2],x^4-6*x^2*y^2+y^4);
? mfcoefs(F,10)
%6 = [0, 4, -16, 0, 64, -56, 0, 0, -256, 324, 224]
? mfcoef(F,100000) \\ instantaneous
%7 = 41304367104
```

Odd dimensions are supported, corresponding to forms of half-integral weight:

```
? [mf,F,v] = mffromqf(2*matid(3));
? mfisequal(F, mfpow(mfTheta(),3))
%2 = 1
? mfcoefs(F, 32) \\ illustrate Legendre's 3-square theorem
%3 = [ 1,
6, 12, 8, 6, 24, 24, 0, 12,
30, 24, 24, 8, 24, 48, 0, 6,
48, 36, 24,24, 48, 24, 0, 24,
30, 72, 32, 0, 72, 48, 0, 12]
```

mfgaloisprojrep(F, precision)

mf being an mf output by mfinit in weight 1, return a polynomial defining the field fixed by the kernel of the projective Artin representation attached to F (by Deligne-Serre). Currently only implemented for projective image A_4 and S_4 .

```
\land A4 example
? mf = mfinit([4*31,1,Mod(87,124)],0);
? F = mfeigenbasis(mf)[1];
? mfgaloistype(mf,F)
\%3 = -12
? pol = mfgaloisprojrep(mf,F)
\%4 = x^{12} + 68^{*}x^{10} + 4808^{*}x^{8} + \ldots + 4096
? G = galoisinit(pol); galoisidentify(G)
\%5 = [12.3] \setminus A4
? pol4 = polredbest(galoisfixedfield(G,G.gen[3], 1))
\%6 = x^4 + 7^*x^2 - 2^*x + 14
? polgalois(pol4)
%7 = [12, 1, 1, "A4"]
? factor(nfdisc(pol4))
%8 =
[24]
[31 2]
\\ S4 example
? mf = mfinit([4*37,1,Mod(105,148)],0);
? F = mfeigenbasis(mf)[1];
? mfgaloistype(mf,F)
\%11 = -24
? pol = mfgaloisprojrep(mf,F)
\%12 = x^{24} + 24^{*}x^{22} + 256^{*}x^{20} + \dots + 255488256
? G = galoisinit(pol); galoisidentify(G)
```

```
%13 = [24, 12] \\S4
? pol4 = polredbest(galoisfixedfield(G,G.gen[3..4], 1))
%14 = x^4 - x^3 + 5*x^2 - 7*x + 12
? polgalois(pol4)
%15 = [24, -1, 1, "S4"]
? factor(nfdisc(pol4))
%16 =
[ 2 2]
[37 3]
```

mfgaloistype(F)

NK being either [N, 1, CHI] or an mf output by mfinit in weight 1, gives the vector of types of Galois representations attached to each cuspidal eigenform, unless the modular form F is specified, in which case only for F (note that it is not tested whether F belongs to the correct modular form space, nor whether it is a cuspidal eigenform). Types A_4 , S_4 , A_5 are represented by minus their cardinality -12, -24, or -60, and type D_n is represented by its cardinality, the integer 2n:

```
? mfgaloistype([124,1, Mod(67,124)]) \\ A4
%1 = [-12]
? mfgaloistype([148,1, Mod(105,148)]) \\ S4
%2 = [-24]
? mfgaloistype([633,1, Mod(71,633)]) \\ D10, A5
%3 = [10, -60]
? mfgaloistype([239,1, -239]) \\ D6, D10, D30
%4 = [6, 10, 30]
? mfgaloistype([71,1, -71])
%5 = [14]
? mf = mfinit([239,1, -239],0); F = mfeigenbasis(mf)[2];
? mfgaloistype(mf, F)
%7 = 10
```

The function may also return 0 as a type when it failed to determine it; in this case the correct type is either -12 or -60, and most likely -12.

mfhecke(F, n)

F being a modular form in modular form space mf, returns T(n)F, where T(n) is the n-th Hecke operator.

Warning. If F is of level M < N, then T(n)F is in general not the same in $M_k(\Gamma_0(M), \chi)$ and in $M_k(\Gamma_0(N), \chi)$. We take T(n) at the same level as the one used in mf.

? mf = mfinit([26,2],0); F = mfbasis(mf)[1]; mftobasis(mf,F) %1 = [1, 0]~ ? G2 = mfhecke(mf,F,2); mftobasis(mf,G2) %2 = [0, 1]~ ? G5 = mfhecke(mf,F,5); mftobasis(mf,G5) %3 = [-2, 1]~

Modular forms of half-integral weight are supported, in which case n must be a perfect square, else T_n will act as 0 (the operator T_p for p || N is not supported yet):

```
? F = mfpow(mfTheta(),3); mf = mfinit(F);
? mfisequal(mfhecke(mf,F,9), mflinear([F],[4]))
%2 = 1
```

(F is an eigenvector of all T_{p^2} , with eigenvalue p + 1 for odd p.)

Warning. When n is a large composite, resp. the square of a large composite in half-integral weight, it is in general more efficient to use mfheckemat on the mftobasis coefficients:

```
? mfcoefs(mfhecke(mf,F,3^10), 10)
time = 917 ms.
%3 = [324, 1944, 3888, 2592, 1944, 7776, 7776, 0, 3888, 9720, 7776]
? M = mfheckemat(mf,3^10) \\ instantaneous
%4 =
[324]
? G = mflinear(mf, M*mftobasis(mf,F));
? mfcoefs(G, 10) \\ instantaneous
%6 = [324, 1944, 3888, 2592, 1944, 7776, 7776, 0, 3888, 9720, 7776]
```

mfheckemat(vecn)

If vecn is an integer, matrix of the Hecke operator T(n) on the basis formed by mfbasis(mf). If it is a vector, vector of such matrices, usually faster than calling each one individually.

```
? mf=mfinit([32,4],0); mfheckemat(mf,3)
%1 =
[0 44 0]
[1 0 -10]
[0 -2 0]
? mfheckemat(mf,[5,7])
%2 = [[0, 0, 220; 0, -10, 0; 1, 0, 12], [0, 88, 0; 2, 0, -20; 0, -4, 0]]
```

mfinit(space)

Create the space of modular forms corresponding to the data contained in NK and space. NK is a vector which can be either [N, k] (N level, k weight) corresponding to a subspace of $M_k(\Gamma_0(N))$, or [N, k, CHI] (CHI a character) corresponding to a subspace of $M_k(\Gamma_0(N), \chi)$. Alternatively, it can be a modular form F or modular form space, in which case we use mfparams to define the space parameters.

The subspace is described by the small integer space: 0 for the newspace $S_k^{new}(\Gamma_0(N), \chi)$, 1 for the cuspidal space S_k , 2 for the oldspace S_k^{old} , 3 for the space of Eisenstein series E_k and 4 for the full space M_k .

Wildcards. For given level and weight, it is advantageous to compute simultaneously spaces attached to different Galois orbits of characters, especially in weight 1. The parameter *CHI* may be set to 0 (wildcard), in which case we return a vector of all mfinit (s) of non trivial spaces in $S_k(\Gamma_1(N))$, one for each Galois orbit (see znchargalois). One may also set *CHI* to a vector of characters and we return a vector of all mfinits of subspaces of $M_k(G_0(N), \chi)$ for χ in the list, in the same order. In weight 1, only S_1^{new} , S_1 and E_1 support wildcards.

The output is a technical structure S, or a vector of structures if *CHI* was a wildcard, which contains the following information: $[N, k, \chi]$ is given by mfparams(S), the space dimension is mfdim(S) and a \mathbb{C} -basis for the space is mfbasis(S). The structure is entirely algebraic and does not depend on the current realbitprecision.

```
? S = mfinit([36,2], 0); \\ new space
? mfdim(S)
%2 = 1
? mfparams
%3 = [36, 2, 1, y] \\ trivial character
? f = mfbasis(S)[1]; mfcoefs(f,10)
%4 = [0, 1, 0, 0, 0, 0, 0, -4, 0, 0, 0]
```

```
? vS = mfinit([36,2,0],0); \\ with wildcard
? #vS
%6 = 4 \\ 4 non trivial spaces (mod Galois action)
? apply(mfdim,vS)
%7 = [1, 2, 1, 4]
? mfdim([36,2,0], 0)
%8 = [[1, Mod(1, 36), 1, 0], [2, Mod(35, 36), 2, 0], [3, Mod(13, 36), 1, 0],
[6, Mod(11, 36), 4, 0]]
```

mfisCM()

Tests whether the eigenform F is a CM form. The answer is 0 if it is not, and if it is, either the unique negative discriminant of the CM field, or the pair of two negative discriminants of CM fields, this latter case occurring only in weight 1 when the projective image is $D_2 = C_2 x C_2$, i.e., coded 4 by mfgaloistype.

```
? F = mffromell(ellinit([0,1]))[2]; mfisCM(F)
%1 = -3
? mf = mfinit([39,1,-39],0); F=mfeigenbasis(mf)[1]; mfisCM(F)
%2 = Vecsmall([-3, -39])
? mfgaloistype(mf)
%3 = [4]
```

mfisequal(G, lim)

Checks whether the modular forms F and G are equal. If \lim is nonzero, only check equality of the first lim + 1Fourier coefficients and the function then also applies to generalized modular forms.

? D = mfDelta(); F = mfderiv(D); ? G = mfmul(mfEk(2), D); ? mfisequal(F, G) %2 = 1

mfisetaquo(flag)

If the generalized modular form f is a holomorphic eta quotient, return the eta quotient matrix, else return 0. If *flag* is set, also accept meromorphic eta quotients: check whether $f = q^{-\nu(g)}g(q)$ for some eta quotient g; if so, return the eta quotient matrix attached to g, else return 0. See mffrometaquo.

```
? mfisetaquo(mfDelta())
%1 =
[1 24]
? f = mffrometaquo([1,1;23,1]);
? mfisetaquo(f)
%3 =
[ 1 1]
[23 1]
? f = mffrometaquo([1,-24], 1);
? mfisetaquo(f) \\ nonholomorphic
%5 = 0
? mfisetaquo(f,1)
%6 =
[1 -24]
```

mfkohnenbasis()

mf being a cuspidal space of half-integral weight $k \ge 3/2$ with level N and character χ , gives a basis B of the Kohnen +-space of mf as a matrix whose columns are the coefficients of B on the basis of mf. The conductor of either χ or χ .(-4/.) must divide N/4.

In the final example both $\chi = (8/.)$ and $\chi \cdot (-4/.)$ have conductor 8, which does not divide N/4 = 10.

mfkohnenbijection()

mf being a cuspidal space of half-integral weight, returns [mf2,M,K,shi], where M is a matrix giving a Hecke-module isomorphism from the cuspidal space mf2 giving $S_{2k-1}(\Gamma_0(N), \chi^2)$ to the Kohnen +-space $S_k^+(\Gamma_0(4N), \chi)$, K represents a basis B of the Kohnen +-space as a matrix whose columns are the coefficients of B on the basis of mf; shi is a vector of pairs (t_i, n_i) gives the linear combination of Shimura lifts giving M^{-1} : t_i is a squarefree positive integer and n_i is a small nonzero integer.

```
? mf=mfinit([60,5/2],1); [mf2,M,K,shi]=mfkohnenbijection(mf); M
%2 =
[-3 0 5/2 7/2]
[ 1 -1/2 -7 -7]
[ 1 1/2 0 -3]
[ 0 0 5/2 5/2]
? shi
%2 = [[1, 1], [2, 1]]
```

This last command shows that the map giving the bijection is the sum of the Shimura lift with t = 1 and the one with t = 2.

Since it gives a bijection of Hecke modules, this matrix can be used to transport modular form data from the easily computed space of level N and weight 2k - 1 to the more difficult space of level 4N and weight k: matrices of Hecke operators, new space, splitting into eigenspaces and eigenforms. Examples:

```
? K^(-1)*mfheckemat(mf,121)*K /* matrix of T_11^2 on K. Slowish. */
time = 1,280 ms.
%1 =
[ 48 24 24 24]
```

```
[ 0 32 0 -20]
[-48 -72 -40 -72]
[ 0 0 0 52]
? M*mfheckemat(mf2,11)*M^(-1) /* instantaneous via T_11 on S_{2k-1} */
time = 0 \text{ ms.}
%2 =
[ 48 24 24 24]
[ 0 32 0 -20]
[-48 -72 -40 -72]
[00052]
? mf20=mfinit(mf2,0); [mftobasis(mf2,b) | b<-mfbasis(mf20)]</pre>
\%3 = [[0, 0, 1, 0] \sim, [0, 0, 0, 1] \sim]
? F1=M*[0,0,1,0]~
\%4 = [1/2, 1/2, -3/2, -1/2] \sim
? F2=M*[0,0,0,1]~
\%5 = [3/2, 1/2, -9/2, -1/2]
? K*F1
%6 = [1, 0, 0, 1, 1, 0, 0, 1, -3, 0, 0, -3, 0, 0]~
? K*F2
%7 = [3, 0, 0, 3, 1, 0, 0, 1, -9, 0, 0, -3, 0, 0]~
```

This gives a basis of the new space of $S_{5/2}^+(\Gamma_0(60))$ expressed on the initial basis of $S_{5/2}(\Gamma_0(60))$. If we want the eigenforms, we write instead:

```
? BE=mfeigenbasis(mf20);[E1,E2]=apply(x->K*M*mftobasis(mf2,x),BE)
%1 = [[1, 0, 0, 1, 0, 0, 0, 0, -3, 0, 0, 0, 0, 0]~,\
[0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, -3, 0, 0]~
? EI1 = mflinear(mf, E1); EI2=mflinear(mf, E2);
```

These are the two eigenfunctions in the space mf, the first (resp., second) will have Shimura image a multiple of BE[1] (resp., BE[2]). The function mfkohneneigenbasis does this directly.

mfkohneneigenbasis(bij)

mf being a cuspidal space of half-integral weight $k \ge 3/2$ and bij being the output of mfkohnenbijection(mf), outputs a 3-component vector [mf0,BNEW,BEIGEN], where BNEW and BEIGEN are two matrices whose columns are the coefficients of a basis of the Kohnen new space and of the eigenforms on the basis of mf respectively, and mf0 is the corresponding new space of integral weight 2k - 1.

```
? mf=mfinit([44,5/2],1);bij=mfkohnenbijection(mf);
? [mf0,BN,BE]=mfkohneneigenbasis(mf,bij);
? BN~
%2 =
[2 0 0 -2 2 0 -8]
[2 0 0 4 14 0 -32]
? BE~
```

```
%3 = [1 0 0 Mod(y-1, y^2-3) Mod(2*y+1, y^2-3) 0 Mod(-4*y-4, y^2-3)]
? lift(mfcoefs(mf,20)*BE[,1])
%4 = [0, 1, 0, 0, y − 1, 2*y + 1, 0, 0, 0, −4*y − 4, 0, 0,\
-5*y + 3, 0, 0, 0, −6, 0, 0, 0, 7*y + 9]~
```

mflinear(v)

vF being a vector of generalized modular forms and v a vector of coefficients of same length, compute the linear combination of the entries of vF with coefficients v. Note. Use this in particular to subtract two forms F and G (with vF = [F, G] and v = [1, -1]), or to multiply an form by a scalar λ (with vF = [F] and $v = [\lambda]$).

? D = mfDelta(); G = mflinear([D],[-3]); ? mfcoefs(G,4) %2 = [0, -3, 72, -756, 4416]

For user convenience, we allow

- a modular form space mf as a vF argument, which is understood as mfbasis(mf);
- in this case, we also allow a modular form f as v, which is understood as mftobasis(mf, f).

```
? T = mfpow(mfTheta(),7); F = mfShimura(T,-3); \\ Shimura lift for D=-3
? mfcoefs(F,8)
%2 = [-5/9, 280, 9240, 68320, 295960, 875280, 2254560, 4706240, 9471000]
? mf = mfinit(F); G = mflinear(mf,F);
? mfcoefs(G,8)
%4 = [-5/9, 280, 9240, 68320, 295960, 875280, 2254560, 4706240, 9471000]
```

This last construction allows to replace a general modular form by a simpler linear combination of basis functions, which is often more efficient:

```
? T10=mfpow(mfTheta(),10); mfcoef(T10, 10^4) \\ direct evaluation
time = 399 ms.
%5 = 128205250571893636
? mf=mfinit(T10); F=mflinear(mf,T10); \\ instantaneous
? mfcoef(F, 10^4) \\ after linearization
time = 67 ms.
%7 = 128205250571893636
```

mfmanin(precision)

Given the modular symbol FS associated to an eigenform F by mfsymbol (mf,F), computes the even and odd special polynomials as well as the even and odd periods ω^+ and ω^- as a vector $[[P^+, P^-], [\omega^+, \omega^-, r]]$, where $r = \Im(\omega^+\overline{\omega^-})/\langle F, F \rangle$. If F has several embeddings into \mathbb{C} , give the vector of results corresponding to each embedding.

```
? D=mfDelta(); mf=mfinit(D); DS=mfsymbol(mf,D);
? [pols,oms]=mfmanin(DS); pols
%2 = [[4*x^9 - 25*x^7 + 42*x^5 - 25*x^3 + 4*x],\
    [-36*x^10 + 691*x^8 - 2073*x^6 + 2073*x^4 - 691*x^2 + 36]]
? oms
%3 = [0.018538552324740326472516069364750571812,\
    -0.00033105361053212432521308691198949874026*I, 4096/691]
? mf=mfinit([11,2],0); F=mfeigenbasis(mf)[1]; FS=mfsymbol(mf,F);
? [pols,oms]=mfmanin(FS);pols
```

```
%5 = [[0, 0, 0, 1, 1, 0, 0, -1, -1, 0, 0, 0],

[2, 0, 10, 5, -5, -10, -10, -5, 5, 10, 0, -2]]

? oms[3]

%6 = 24/5
```

mfmul(G)

Multiply the two generalized modular forms F and G.

```
? E4 = mfEk(4); G = mfmul(mfmul(E4,E4),E4);
? mfcoefs(G, 4)
%2 = [1, 720, 179280, 16954560, 396974160]
? mfcoefs(mfpow(E4,3), 4)
%3 = [1, 720, 179280, 16954560, 396974160]
```

mfnumcusps()

Number of cusps of $\Gamma_0(N)$

```
? mfnumcusps(24)
%1 = 8
? mfcusps(24)
%1 = [0, 1/2, 1/3, 1/4, 1/6, 1/8, 1/12, 1/24]
```

mfparams()

If F is a modular form space, returns [N,k,CHI,space,:math:Phi`], level, weight, character χ , and space code; where Φ is the cyclotomic polynomial defining the field of values of CHI. If F is a generalized modular form, returns [N,k,CHI,P,:math:Phi`], where P is the (polynomial giving the) field of definition of F as a relative extension of the cyclotomic field $\mathbb{Q}(\chi) = \mathbb{Q}[t]/(\Phi)$: in that case the level N may be a multiple of the level of F and the polynomial P may define a larger field than $\mathbb{Q}(F)$. If you want the true level of F from this result, use mfconductor(mfinit(F),F). The polynomial P defines an extension of $\mathbb{Q}(\chi) = \mathbb{Q}[t]/(\Phi(t))$; it has coefficients in that number field (polmods in t).

In contrast with mfparams(F)[4] which always gives the polynomial P defining the relative extension $\mathbb{Q}(F)/\mathbb{Q}(\chi)$, the member function :math: F.mod` returns the polynomial used to define $\mathbb{Q}(F)$ over \mathbb{Q} (either a cyclotomic polynomial or a polynomial with cyclotomic coefficients).

```
? E1 = mfeisenstein(4,-3,-4); E2 = mfeisenstein(3,5,-7); E3 = mfmul(E1,E2);
? apply(mfparams, [E1,E2,E3])
%2 = [[12, 4, 12, y, t-1], [35, 3, -35, y, t-1], [420, 7, -420, y, t-1]]
? mf = mfinit([36,2,Mod(13,36)],0); [f] = mfeigenbasis(mf); mfparams(mf)
%3 = [36, 2, Mod(13, 36), 0, t^2 + t + 1]
? mfparams(f)
%4 = [36, 2, Mod(13, 36), y, t^2 + t + 1]
? f.mod
%5 = t^2 + t + 1
? mf = mfinit([36,4,Mod(13,36)],0); [f] = mfeigenbasis(mf);
? lift(mfparams(f))
%7 = [36, 4, 13, y^3 + (2*t-2)*y^2 + (-4*t+6)*y + (10*t-1), t^2+t+1]
```

mfperiodpol(f, flag, precision)

Period polynomial of the cuspidal part of the form f, in other words $\int_0^{ioo} (X-\tau)^{k-2} f(\tau) d\tau$. If flag is 0,

ordinary period polynomial. If it is 1 or -1, even or odd part of that polynomial. f can also be the modular symbol output by mfsymbol (mf,f).

? D = mfDelta(); mf = mfinit(D,0); ? PP = mfperiodpol(mf, D, -1); PP/=polcoef(PP, 1); bestappr(PP) %1 = x^9 - 25/4*x^7 + 21/2*x^5 - 25/4*x^3 + x ? PM = mfperiodpol(mf, D, 1); PM/=polcoef(PM, 0); bestappr(PM) %2 = -x^10 + 691/36*x^8 - 691/12*x^6 + 691/12*x^4 - 691/36*x^2 + 1

mfpetersson(gs)

Petersson scalar product of the modular forms f and g belonging to the same modular form space mf, given by the corresponding "modular symbols" fs and gs output by mfsymbol (also in weight 1 and half-integral weight, where symbols do not exist). If gs is omitted it is understood to be equal to fs. The scalar product is normalized by the factor $1/[\Gamma : \Gamma_0(N)]$. Note that f and g can both be noncuspidal, in which case the program returns an error if the product is divergent. If the fields of definition $\mathbb{Q}(f)$ and $\mathbb{Q}(g)$ are equal to $\mathbb{Q}(\chi)$ the result is a scalar. If $[\mathbb{Q}(f) : \mathbb{Q}(\chi)] = d > 1$ and $[\mathbb{Q}(g) : \mathbb{Q}(\chi)] = e > 1$ the result is a dxe matrix corresponding to all the embeddings of f and g. In the intermediate cases d = 1 or e = 1 the result is a row or column vector.

```
? D=mfDelta(); mf=mfinit(D); DS=mfsymbol(mf,D); mfpetersson(DS)
%1 = 1.0353620568043209223478168122251645932 E-6
? mf=mfinit([11,6],0);B=mfeigenbasis(mf);BS=vector(#B,i,mfsymbol(mf,B[i]));
? mfpetersson(BS[1])
%3 = 1.6190120685220988139111708455305245466 E-5
? mfpetersson(BS[1].BS[2])
%4 = [-3.826479006582967148 E-42 - 2.801547395385577002 E-41*I,\
1.6661127341163336125 E-41 + 1.1734725972345985061 E-41*I,
0.E-42 - 6.352626992842664490 E-41*I]~
? mfpetersson(BS[2])
%5 =
[ 2.7576133733... E-5 2.0... E-42 6.3... E-43 ]
[ -4.1... E-42 6.77837030070... E-5 3.3...E-42 ]
[-6.32...E-43 3.6... E-42 2.27268958069... E-5]
? mf=mfinit([23,2],0); F=mfeigenbasis(mf)[1]; FS=mfsymbol(mf,F);
? mfpetersson(FS)
%5 =
[0.0039488965740025031688548076498662860143 -3.56 ... E-40]
[-3.5... E-40 0.0056442542987647835101583821368582485396]
```

Noncuspidal example:

```
? E1=mfeisenstein(5,1,-3);E2=mfeisenstein(5,-3,1);
? mf=mfinit([12,5,-3]); cusps=mfcusps(12);
? apply(x->mfcuspval(mf,E1,x),cusps)
%3 = [0, 0, 1, 0, 1, 1]
? apply(x->mfcuspval(mf,E2,x),cusps)
%4 = [1/3, 1/3, 0, 1/3, 0, 0]
? E1S=mfsymbol(mf,E1);E2S=mfsymbol(mf,E2);
? mfpetersson(E1S,E2S)
%6 = -1.884821671646... E-5 - 1.9... E-43*I
```

Weight 1 and 1/2-integral weight example:

```
? mf=mfinit([23,1,-23],1);F=mfbasis(mf)[1];FS=mfsymbol(mf,F);
? mfpetersson(mf,FS)
%2 = 0.035149946790370230814006345508484787443
? mf=mfinit([4,9/2],1);F=mfbasis(mf)[1];FS=mfsymbol(mf,F);
? mfpetersson(FS)
%4 = 0.00015577084407139192774373662467908966030
```

mfpow(n)

Compute F^n , where n is an integer and F is a generalized modular form:

```
? G = mfpow(mfEk(4), 3); \\ E4^3
? mfcoefs(G, 4)
%2 = [1, 720, 179280, 16954560, 396974160]
```

mfsearch(V, space)

NK being of the form [N,k] with k possibly half-integral, search for a modular form with rational coefficients, of weight k and level N, whose initial coefficients a(0),... are equal to V; space specifies the modular form spaces in which to search, in mfinit or mfdim notation. The output is a list of matching forms with that given level and weight. Note that the character is of the form (D/.), where D is a (positive or negative) fundamental discriminant dividing N. The forms are sorted by increasing ||D||.

The parameter N can be replaced by a vector of allowed levels, in which case the list of forms is sorted by increasing level, then increasing ||D||. If a form is found at level N, any multiple of N with the same D is not considered. Some useful possibilities are

- [:math:`N_1..:math:N_2]`: all levels between N₁ and N₂, endpoints included;
- :math: F * [N₁..:math:N_2]`: same but levels divisible by F;
- divisors (N_0) : all levels dividing N_0 .

Note that this is different from mfeigensearch, which only searches for rational eigenforms.

```
? F = mfsearch([[1..40], 2], [0,1,2,3,4], 1); #F
%1 = 3
? [ mfparams(f)[1..3] | f <- F ]
%2 = [[38, 2, 1], [40, 2, 8], [40, 2, 40]]
? mfcoefs(F[1],10)
%3 = [0, 1, 2, 3, 4, -5, -8, 1, -7, -5, 7]
```

mfshift(s)

Divide the generalized modular form F by q^s , omitting the remainder if there is one. One can have s < 0.

```
? D=mfDelta(); mfcoefs(mfshift(D,1), 4)
%1 = [1, -24, 252, -1472, 4830]
? mfcoefs(mfshift(D,2), 4)
%2 = [-24, 252, -1472, 4830, -6048]
? mfcoefs(mfshift(D,-1), 4)
%3 = [0, 0, 1, -24, 252]
```

mfshimura(F, D)

F being a modular form of half-integral weight $k \ge 3/2$ and t a positive squarefree integer, returns the Shimura lift G of weight 2k - 1 corresponding to D. This function returns [mf2, G, v] where mf2 is a modular form space containing G and v expresses G in terms of mfbasis(mf2); so that G is mflinear(mf2, v).

```
? F = mfpow(mfTheta(), 7); mf = mfinit(F);
? [mf2, G, v] = mfshimura(mf, F, 3); mfcoefs(G,5)
\%2 = [-5/9, 280, 9240, 68320, 295960, 875280]
? mfparams(G) \\ the level may be lower than expected
\%3 = [1, 6, 1, y, t - 1]
? mfparams(mf2)
\%4 = [2, 6, 1, 4, t - 1]
? v
\%5 = [280, 0] \sim
? mfcoefs(mf2, 5)
%6 =
[-1/504 - 1/504]
[ 1 0]
[ 33 1]
[ 244 0]
[ 1057 33]
[ 3126 0]
? mf = mfinit([60,5/2],1); F = mflinear(mf,mfkohnenbasis(mf)[,1]);
? mfparams(mfshimura(mf,F)[2])
\%8 = [15, 4, 1, y, t - 1]
? mfparams(mfshimura(mf,F,6)[2])
\%9 = [15, 4, 1, y, t - 1]
```

mfslashexpansion(*f*, *g*, *n*, *flrat*, *params*, *precision*)

Let *mf* be a modular form space in level N, f a modular form belonging to *mf* and let g be in $M_2^+(Q)$. This function computes the Fourier expansion of $f||_k g$ to n terms. We first describe the behaviour when flrat is 0: the result is a vector v of floating point complex numbers such that

$$f||_k g(\tau) = q^{\alpha} \sum_{m>=0} v[m+1]q^{m/w},$$

where $q = e(\tau)$, w is the width of the cusp g(ioo) (namely $(N/(c^2, N)$ if g is integral) and α is a rational number. If params is given, it is set to the parameters $[\alpha, w, matid(2)]$.

If flrat is 1, the program tries to rationalize the expression, i.e., to express the coefficients as rational numbers or polmods. We write $g = \lambda.M.A$ where $\lambda \in \mathbb{Q}^*$, $M \in SL_2(\mathbb{Z})$ and A = [a, b; 0, d] is upper triangular, integral and primitive with a > 0, d > 0 and $0 \le b \le d$. Let α and w by the parameters attached to the expansion of $F := f \parallel_k M$ as above, i.e.

$$F(\tau) = q^{\alpha} \sum_{m>=0} v[m+1]q^{m/w}.$$

The function returns the expansion v of $F = f \parallel_k M$ and sets the parameters to $[\alpha, w, A]$. Finally, the desired expansion is $(a/d)^{k/2}F(\tau + b/d)$. The latter is identical to the returned expansion when A is the identity, i.e. when $g \in PSL_2(\mathbb{Z})$. If this is not the case, the expansion differs from v by the multiplicative constant $(a/d)^{k/2}e(\alpha b/(dw))$ and a twist by a root of unity $q^{1/w} \to e(b/(dw))q^{1/w}$. The complications introduced by this extra matrix A allow to recognize the coefficients in a much smaller cyclotomic field, hence to obtain a simpler description overall. (Note that this rationalization step may result in an error if the program cannot perform it.)

```
? mf = mfinit([32,4],0); f = mfbasis(mf)[1];
? mfcoefs(f, 10)
\%2 = [0, 3, 0, 0, 0, 2, 0, 0, 0, 47, 0]
? mfatk = mfatkininit(mf,32); mfcoefs(mfatkin(mfatk,f),10) / mfatk[3]
\%3 = [0, 1, 0, 16, 0, 22, 0, 32, 0, -27, 0]
? mfatk[3] \ here normalizing constant C = 1, but need in general
\%4 = 1
? mfslashexpansion(mf,f,[0,-1;1,0],10,1,&params) * 32^(4/2)
\%5 = [0, 1, 0, 16, 0, 22, 0, 32, 0, -27, 0]
? params
\%6 = [0, 32, [1, 0; 0, 1]]
? mf = mfinit([12,8],0); f = mfbasis(mf)[1];
? mfslashexpansion(mf, f, [1,0;2,1],7,0)
%7 = [0, 0, 0, 0.66666666... + 0.E-38*I, 0, −3.9999999... + 6.92820...*I, 0,\
-11.99999999... - 20.78460969...*I]
? mfslashexpansion(mf,f,[1,0;2,1],7,1, &params)
%8 = [0, 0, 0, 2/3, 0, Mod(8*t, t^2+t+1), 0, Mod(-24*t-24, t^2+t+1)]
? params
\%9 = [0, 3, [1, 0; 0, 1]]
```

If $[\mathbb{Q}(f) : \mathbb{Q}(\chi)] > 1$, the coefficients may be polynomials in y, where y is any root of the polynomial giving the field of definition of f (f.mod or mfparams(f)[4]).

```
? mf=mfinit([23,2],0);f=mfeigenbasis(mf)[1];
? mfcoefs(f,5)
%1 = [Mod(0, y^2 - y - 1), Mod(1, y^2 - y - 1), Mod(-y, y^2 - y - 1),
Mod(2*y - 1, y^2 - y - 1), Mod(y - 1, y^2 - y - 1), Mod(-2*y, y^2 - y - 1)]
? mfslashexpansion(mf,f,[1,0;0,1],5,1)
%2 = [0, 1, -y, 2*y - 1, y - 1, -2*y]
? mfslashexpansion(mf,f,[0,-1;1,0],5,1)
%3 = [0, -1/23, 1/23*y, -2/23*y + 1/23, -1/23*y + 1/23, 2/23*y]
```

Caveat. In half-integral weight, we define the "slash" operation as

$$(f||_k g)(\tau) := ((c\tau + d)^{-1/2})^{2k} f(g.\tau),$$

with the principal determination of the square root. In particular, the standard cocycle condition is no longer satisfied and we only have f ||(gg') = (f||g)||g'.

mfspace(*f*)

Identify the modular space mf, resp. the modular form f in mf if present, as the flag given to mfinit. Returns 0 (newspace), 1 (cuspidal space), 2 (old space), 3 (Eisenstein space) or 4 (full space).

? mf = mfinit([1,12],1); mfspace(mf)
%1 = 1
? mfspace(mf, mfDelta())
%2 = 0 \\ new space

This function returns -1 when the form f is modular but does not belong to the space.

```
? mf = mfinit([1,2]; mfspace(mf, mfEk(2))
%3 = -1
```

When *f* is not modular and is for instance only quasi-modular, the function returns nonsense:

```
? M6 = mfinit([1,6]);
? dE4 = mfderiv(mfEk(4)); \\ not modular !
? mfspace(M6,dE4) \\ asserts (wrongly) that E4' belongs to new space
%3 = 0
```

mfsplit(dimlim, flag)

mf from mfinit with integral weight containing the new space (either the new space itself or the cuspidal space or the full space), and preferably the newspace itself for efficiency, split the space into Galois orbits of eigenforms of the newspace, satisfying various restrictions.

The functions returns [vF, vK], where vF gives (Galois orbit of) eigenforms and vK is a list of polynomials defining each Galois orbit. The eigenforms are given in mftobasis format, i.e. as a matrix whose columns give the forms with respect to mfbasis(mf).

If dimlim is set, only the Galois orbits of dimension $\langle = dimlim$ are computed (i.e. the rational eigenforms if dimlim = 1 and the character is real). This can considerably speed up the function when a Galois orbit is defined over a large field.

flag speeds up computations when the dimension is large: if flag = d > 0, when the dimension of the eigenspace is > d, only the Galois polynomial is computed.

Note that the function mfeigenbasis returns all eigenforms in an easier to use format (as modular forms which can be input as is in other functions); mfsplit is only useful when you can restrict to orbits of small dimensions, e.g. rational eigenforms.

```
? mf=mfinit([11,2],0); f=mfeigenbasis(mf)[1]; mfcoefs(f,16)
\%1 = [0, 1, -2, -1, \ldots]
? mf=mfinit([23,2],0); f=mfeigenbasis(mf)[1]; mfcoefs(f,16)
%2 = [Mod(0, z^2 - z - 1), Mod(1, z^2 - z - 1), Mod(-z, z^2 - z - 1), ...]
? mf=mfinit([179,2],0); apply(poldegree, mffields(mf))
\%3 = [1, 3, 11]
? mf=mfinit([719,2],0);
? [vF,vK] = mfsplit(mf, 5); \\ fast when restricting to small orbits
time = 192 \text{ ms.}
? #vF \setminus a single orbit
%5 = 1
? poldegree(vK[1]) \\ of dimension 5
\%6 = 5
? [vF,vK] = mfsplit(mf); \\ general case is slow
time = 2,104 ms.
? apply(poldegree,vK)
\%8 = [5, 10, 45] \setminus because degree 45 is large...
```

mfsturm()

Gives the Sturm bound for modular forms on $\Gamma_0(N)$ and weight k, i.e., an upper bound for the order of the zero at infinity of a nonzero form. NK is either

- a pair [N, k], in which case the bound is the floor of (kN/12). $\prod_{p \parallel N} (1 + 1/p)$;
- or the output of mfinit in which case the exact upper bound is returned.

```
? NK = [96,6]; mfsturm(NK)
%1 = 97
? mf=mfinit(NK,1); mfsturm(mf)
%2 = 76
```

? mfdim(NK,0) \\ new space %3 = 72

mfsymbol(f, precision)

Initialize data for working with all period polynomials of the modular form f: this is essential for efficiency for functions such as mfsymboleval, mfmanin, and mfpetersson. An mfsymbol contains an mf structure and can always be used whenever an mf would be needed.

```
? mf=mfinit([23,2],0);F=mfeigenbasis(mf)[1];
? FS=mfsymbol(mf,F);
? mfsymboleval(FS,[0,oo])
%3 = [8.762565143790690142 E-39 + 0.0877907874...*I,
-5.617375463602574564 E-39 + 0.0716801031...*I]
? mfpetersson(FS)
%4 =
[0.0039488965740025031688548076498662860143 1.2789721111175127425 E-40]
[1.2630501762985554269 E-40 0.0056442542987647835101583821368582485396]
```

By abuse of language, initialize data for working with mfpetersson in weight 1 and half-integral weight (where no symbol exist); the mf argument may be an mfsymbol attached to a form on the space, which avoids recomputing data independent of the form.

```
? mf=mfinit([12,9/2],1); F=mfbasis(mf);
? fs=mfsymbol(mf,F[1]);
time = 476 ms
? mfpetersson(fs)
%2 = 1.9722437519492014682047692073275406145 E-5
? f2s = mfsymbol(mf,F[2]);
time = 484 ms.
? mfpetersson(f2s)
%4 = 1.2142222531326333658647877864573002476 E-5
? gs = mfsymbol(fs,F[2]); \\ re-use existing symbol, a little faster
time = 430 ms.
? mfpetersson(gs) == %4 \\ same value
%6 = 1
```

For simplicity, we also allow mfsymbol(f) instead of mfsymbol(mfinit(f), f):

mfsymboleval(path, ga, precision)

Evaluation of the modular symbol fs (corresponding to the modular form f) output by mfsymbol on the given path path, where path is either a vector $[s_1, s_2]$ or an integral matrix [a, b; c, d] representing the path [a/c, b/d]. In both cases s_1 or s_2 (or a/c or b/d) can also be elements of the upper half-plane. To avoid possibly lengthy mfsymbol computations, the program also accepts fs of the form [mf, F], but in that case s_1 and s_2 are limited to oo and elements of the upper half-plane. The result is the polynomial equal to $\int_{s_1}^{s_2} (X - \tau)^{k-2} F(\tau) d\tau$, the integral being computed along a geodesic joining s_1 and s_2 . If ga in $GL_2^+(\mathbb{Q})$ is given, replace F by $F||_k\gamma$. Note that if the integral diverges, the result will be a rational function. If the field of definition $\mathbb{Q}(f)$ is larger than $\mathbb{Q}(\chi)$ then f can be embedded into \mathbb{C} in $d = [\mathbb{Q}(f) : \mathbb{Q}(\chi)]$ ways, in which case a vector of the d results is returned.

```
? mf=mfinit([35,2],1);f=mfbasis(mf)[1];fs=mfsymbol(mf,f);
? mfsymboleval(fs,[0,oo])
%1 = 0.31404011074188471664161704390256378537*I
```

```
? mfsymboleval(fs,[1,3;2,5])
%2 = -0.1429696291... - 0.2619975641...*I
? mfsymboleval(fs,[I,2*I])
%3 = 0.00088969563028739893631700037491116258378*I
? E2=mfEk(2);E22=mflinear([E2,mfbd(E2,2)],[1,-2]);mf=mfinit(E22);
? E2S = mfsymbol(mf,E22);
? mfsymboleval(E2S,[0,1])
%6 = (-1.00000...*x^2 + 1.00000...*x - 0.50000...)/(x^2 - x)
```

The rational function which is given in case the integral diverges is easy to interpret. For instance:

? E4=mfEk(4);mf=mfinit(E4);ES=mfsymbol(mf,E4); ? mfsymboleval(ES,[I,oo]) %2 = 1/3*x^3 - 0.928067...*I*x^2 - 0.833333...*x + 0.234978...*I ? mfsymboleval(ES,[0,I]) %3 = (-0.234978...*I*x^3 - 0.833333...*x^2 + 0.928067...*I*x + 0.333333...)/x

<code>mfsymboleval(ES,[a,oo])</code> is the limit as $T \rightarrow oo$ of

$$\int_{a}^{iT} (X-\tau)^{k-2} F(\tau) d\tau + a(0)(X-iT)^{k-1}/(k-1),$$

where a(0) is the 0 at infinity. Similarly, mfsymboleval(ES,[0,a]) is the limit as $T \to oo$ of

$$\int_{i/T}^{a} (X-\tau)^{k-2} F(\tau) d\tau + b(0)(1+iTX)^{k-1}/(k-1),$$

where b(0) is the 0 at infinity.

mftaylor(*n*, *flreal*, *precision*)

F being a form in $M_k(SL_2(\mathbb{Z}))$, computes the first n + 1 canonical Taylor expansion of F around $\tau = I$. If flreal = 0, computes only an algebraic equivalence class. If flreal is set, compute p_n such that for τ close enough to I we have

$$f(\tau) = (2I/(\tau+I))^k \sum_{n>=0} p_n((\tau-I)/(\tau+I))^n.$$

? D=mfDelta(); ? mftaylor(D,8) %2 = [1/1728, 0, -1/20736, 0, 1/165888, 0, 1/497664, 0, -11/3981312]

mftobasis(F, flag)

Coefficients of the form F on the basis given by mfbasis(mf). A q-expansion or vector of coefficients can also be given instead of F, but in this case an error message may occur if the expansion is too short. An error message is also given if F does not belong to the modular form space. If flag is set, instead of error messages the output is an affine space of solutions if a q-expansion or vector of coefficients is given, or the empty column otherwise.

```
? mf = mfinit([26,2],0); mfdim(mf)
%1 = 2
? F = mflinear(mf,[a,b]); mftobasis(mf,F)
%2 = [a, b]~
```

A q-expansion or vector of coefficients can also be given instead of F.

```
? Th = 1 + 2*sum(n=1, 8, q^(n^2), 0(q^80));
? mf = mfinit([4,5,Mod(3,4)]);
? mftobasis(mf, Th^10)
%3 = [64/5, 4/5, 32/5]~
```

If F does not belong to the corresponding space, the result is incorrect and simply matches the coefficients of F up to some bound, and the function may either return an empty column or an error message. If flag is set, there are no error messages, and the result is an empty column if F is a modular form; if F is supplied via a series or vector of coefficients which does not contain enough information to force a unique (potential) solution, the function returns [v, K] where v is a solution and K is a matrix of maximal rank describing the affine space of potential solutions v + K.x.

```
? mf = mfinit([4,12],1);
? mftobasis(mf, q-24*q^2+0(q^3), 1)
%2 = [[43/64, -63/8, 800, 21/64]~, [1, 0; 24, 0; 2048, 768; -1, 0]]
? mftobasis(mf, [0,1,-24,252], 1)
\%3 = [[1, 0, 1472, 0] \sim, [0; 0; 768; 0]]
? mftobasis(mf, [0,1,-24,252,-1472], 1)
\%4 = [1, 0, 0, 0] \sim \setminus  now uniquely determined
? mftobasis(mf, [0,1,-24,252,-1472,0], 1)
\%5 = [1, 0, 0, 0] \sim \setminus  wrong result: no such form exists
? mfcoefs(mflinear(mf,%), 5) \\ double check
\%6 = [0, 1, -24, 252, -1472, 4830]
? mftobasis(mf, [0,1,-24,252,-1472,0])
 *** at top-level: mftobasis(mf,[0,1,
 *** mftobasis: domain error in mftobasis: form does not belong to space
? mftobasis(mf, mfEk(10))
 *** at top-level: mftobasis(mf,mfEk(
 *** ^_____
*** mftobasis: domain error in mftobasis: form does not belong to space
? mftobasis(mf, mfEk(10), 1)
%7 = []~
```

mftonew(F)

mf being being a full or cuspidal space with parameters $[N, k, \chi]$ and F a cusp form in that space, returns a vector of 3-component vectors [M, d, G], where $f(\chi) ||M||N, d||N/M$, and G is a form in $S_k^{new}(\Gamma_0(M), \chi)$ such that F is equal to the sum of the B(d)(G) over all these 3-component vectors.

```
? mf = mfinit([96,6],1); F = mfbasis(mf)[60]; s = mftonew(mf,F); #s
%1 = 1
? [M,d,G] = s[1]; [M,d]
%2 = [48, 2]
? mfcoefs(F,10)
%3 = [0, 0, -160, 0, 0, 0, 0, 0, 0, 0, -14400]
? mfcoefs(G,10)
%4 = [0, 0, -160, 0, 0, 0, 0, 0, 0, 0, -14400]
```

mftraceform(space)

If NK = [N, k, CHI, .] as in mfinit with k integral, gives the trace form in the corresponding subspace of $S_k(\Gamma_0(N), \chi)$. The supported values for space are 0: the newspace (default), 1: the full cuspidal space.

? F = mftraceform([23,2]); mfcoefs(F,16) %1 = [0, 2, -1, 0, -1, -2, -5, 2, 0, 4, 6, -6, 5, 6, 4, -10, -3] ? F = mftraceform([23,1,-23]); mfcoefs(F,16) %2 = [0, 1, -1, -1, 0, 0, 1, 0, 1, 0, 0, 0, 0, -1, 0, 0, -1]

mftwist(D)

F being a generalized modular form, returns the twist of F by the integer D, i.e., the form G such that mfcoef(G, n) = `:math:`(D/n)mfcoef(F,n), where (D/n) is the Kronecker symbol.

```
? mf = mfinit([11,2],0); F = mfbasis(mf)[1]; mfcoefs(F, 5)
%1 = [0, 1, -2, -1, 2, 1]
? G = mftwist(F,-3); mfcoefs(G, 5)
%2 = [0, 1, 2, 0, 2, -1]
? mf2 = mfinit([99,2],0); mftobasis(mf2, G)
%3 = [1/3, 0, 1/3, 0]~
```

Note that twisting multiplies the level by D^2 . In particular it is not an involution:

```
? H = mftwist(G,-3); mfcoefs(H, 5)
%4 = [0, 1, -2, 0, 2, 1]
? mfparams(G)
%5 = [99, 2, 1, y, t - 1]
```

$\min(y)$

Creates the maximum of x and y when they can be compared.

minpoly(v)

minimal polynomial of A with respect to the variable v., i.e. the monic polynomial P of minimal degree (in the variable v) such that P(A) = 0.

modreverse()

Let z = Mod(A, T) be a polmod, and Q be its minimal polynomial, which must satisfy deg(Q) = deg(T). Returns a "reverse polmod" Mod(B, Q), which is a root of T.

This is quite useful when one changes the generating element in algebraic extensions:

? u = Mod(x, x³ - x -1); v = u⁵; ? w = modreverse(v) %2 = Mod(x² - 4*x + 1, x³ - 5*x² + 4*x - 1)

which means that $x^3 - 5x^2 + 4x - 1$ is another defining polynomial for the cubic field

$$\mathbb{Q}(u) = \mathbb{Q}[x]/(x^3 - x - 1) = \mathbb{Q}[x]/(x^3 - 5x^2 + 4x - 1) = \mathbb{Q}(v),$$

and that $u \to v^2 - 4v + 1$ gives an explicit isomorphism. From this, it is easy to convert elements between the $A(u) \in \mathbb{Q}(u)$ and $B(v) \in \mathbb{Q}(v)$ representations:

? A = u² + 2*u + 3; subst(lift(A), 'x, w) %3 = Mod(x² - 3*x + 3, x³ - 5*x² + 4*x - 1) ? B = v² + v + 1; subst(lift(B), 'x, v) %4 = Mod(26*x² + 31*x + 26, x³ - x - 1)

If the minimal polynomial of z has lower degree than expected, the routine fails
```
? u = Mod(-x^3 + 9*x, x^4 - 10*x^2 + 1)
? modreverse(u)
*** modreverse: domain error in modreverse: deg(minpoly(z)) < 4
*** Break loop: type 'break' to go back to GP prompt
break> Vec( dbg_err() ) \\ ask for more info
["e_DOMAIN", "modreverse", "deg(minpoly(z))", "<", 4,
Mod(-x^3 + 9*x, x^4 - 10*x^2 + 1)]
break> minpoly(u)
x^2 - 8
```

moebius()

Moebius μ -function of ||x||; x must be a nonzero integer.

msatkinlehner(Q, H)

Let M be a full modular symbol space of level N, as given by msinit, let Q||N, (Q, N/Q) = 1, and let H be a subspace stable under the Atkin-Lehner involution w_Q . Return the matrix of w_Q acting on H (M if omitted).

```
? M = msinit(36,2); \\ M_2(Gamma_0(36))
? w = msatkinlehner(M,4); w^2 == 1
%2 = 1
? #w \\ involution acts on a 13-dimensional space
%3 = 13
? M = msinit(36,2, -1); \\ M_2(Gamma_0(36))^-
? w = msatkinlehner(M,4); w^2 == 1
%5 = 1
? #w
%6 = 4
```

mscosets(inH)

gen being a system of generators for a group G and H being a subgroup of finite index in G, return a list of right cosets of H and the right action of G on H. The subgroup H is given by a criterion inH (closure) deciding whether an element of G belongs to H. The group G is restricted to types handled by generic multiplication (*) and inversion (g^(-1)), such as matrix groups or permutation groups.

Let $gens = [g_1, ..., g_r]$. The function returns [C, M] where C lists the h = [G : H] representatives $[\gamma_1, ..., \gamma_h]$ for the right cosets $H\gamma_1, ..., H\gamma_h$; γ_1 is always the neutral element in G. For all $i \le h, j \le r$, if M[i][j] = k then $H\gamma_ig_j = H\gamma_k$.

```
? PSL2 = [[0,1;-1,0], [1,1;0,1]]; \\ S and T
\\ G = PSL2, H = Gamma0(2)
? [C, M] = mscosets(PSL2, g->g[2,1] % 2 == 0);
? C \\ three cosets
%3 = [[1, 0; 0, 1], [0, 1; -1, 0], [0, 1; -1, -1]]
? M
%4 = [Vecsmall([2, 1]), Vecsmall([1, 3]), Vecsmall([3, 2])]
```

Looking at M[1] we see that S belongs to the second coset and T to the first (trivial) coset.

mscuspidal(flag)

M being a full modular symbol space, as given by msinit, return its cuspidal part *S*. If flag = 1, return [S, E] its decomposition into cuspidal and Eisenstein parts.

A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

```
? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+
? [S,E] = mscuspidal(M, 1);
? E[1] \\ 2-dimensional
%3 =
[0 -10]
[0 -15]
[0 -3]
[1 0]
? S[1] \\ 1-dimensional
%4 =
[ 3]
[30]
[ 6]
[-8]
```

msdim()

M being a full modular symbol space or subspace, for instance as given by msinit or mscuspidal, return its dimension as a \mathbb{Q} -vector space.

? M = msinit(11,4); msdim(M)
%1 = 6
? M = msinit(11,4,1); msdim(M)
%2 = 4 \\ dimension of the '+' part
? [S,E] = mscuspidal(M,1);
? [msdim(S), msdim(E)]
%4 = [2, 2]

Note that mfdim([N,k]) is going to be much faster if you only need the dimension of the space and not really to work with it. This function is only useful to quickly check the dimension of an existing space.

mseisenstein()

M being a full modular symbol space, as given by msinit, return its Eisenstein subspace. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace. This is the same basis as given by the second component of mscuspidal(M, 1).

```
? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+
? E = mseisenstein(M);
? E[1] \\ 2-dimensional
%3 =
[0 -10]
[0 -15]
[0 -3]
```

[1 0]
? E == mscuspidal(M,1)[2]
%4 = 1

mseval(s, p)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M be a full modular symbol space, as given by msinit, let s be a modular symbol from M, i.e. an element of $\operatorname{Hom}_G(\Delta_0, V)$, and let $p = [a, b] \in \Delta_0$ be a path between two elements in $\mathbb{P}^1(\mathbb{Q})$, return $s(p) \in V$. The path extremities a and b may be given as t_INT, t_FRAC or oo = (1 : 0); it is also possible to describe the path by a $2x^2$ integral matrix whose columns give the two cusps. The symbol s is either

- a t_COL coding a modular symbol in terms of the fixed basis of Hom_G(Δ₀, V) chosen in M; if M was initialized with a nonzero sign (+ or −), then either the basis for the full symbol space or the -part can be used (the dimension being used to distinguish the two).
- a t_MAT whose columns encode modular symbols as above. This is much faster than evaluating individual symbols on the same path p independently.
- a t_VEC (v_i) of elements of V, where the $v_i = s(g_i)$ give the image of the generators g_i of Δ_0 , see mspathgens. We assume that s is a proper symbol, i.e. that the v_i satisfy the mspathgens relations.

If p is omitted, convert a single symbol s to the second form: a vector of the $s(g_i)$. A t_MAT is converted to a vector of such.

```
? M = msinit(2,8,1); \\ M_8(Gamma_0(2))^+
? g = mspathgens(M)[1]
%2 = [[+oo, 0], [0, 1]]
? N = msnew(M)[1]; #N \\ Q-basis of new subspace, dimension 1
%3 = 1
? s = N[,1] \\ t_COL representation
%4 = [-3, 6, -8]~
? S = mseval(M, s) \\ t_VEC representation
%5 = [64*x^6-272*x^4+136*x^2-8, 384*x^5+960*x^4+192*x^3-672*x^2-432*x-72]
? mseval(M,s, g[1])
%6 = 64*x^6 - 272*x^4 + 136*x^2 - 8
? mseval(M,S, g[1])
%7 = 64*x^6 - 272*x^4 + 136*x^2 - 8
```

Note that the symbol should have values in $V = \mathbb{Q}[x, y]_{k-2}$, we return the de-homogenized values corresponding to y = 1 instead.

msfarey(inH, CM)

F being a Farey symbol attached to a group G contained in $PSL_2(\mathbb{Z})$ and H a subgroup of G, return a Farey symbol attached to H. The subgroup H is given by a criterion inH (closure) deciding whether an element of G belongs to H. The symbol F can be created using

- mspolygon: $G = \Gamma_0(N)$, which runs in time O(N);
- or msfarey itself, which runs in time $O([G:H]^2)$.

If present, the argument CM is set to mscosets(F[3]), giving the right cosets of H and the action of G by right multiplication. Since msfarey's algorithm is quadratic in the index [G : H], it is advisable to construct subgroups by a chain of inclusions if possible.

```
\\ Gamma_0(N)
G0(N) = mspolygon(N);
```

```
\\ Gamma_1(N): direct construction, slow
G1(N) = msfarey(mspolygon(1), g -> my(a = g[1,1]%N, c = g[2,1]%N);\
c == 0 && (a == 1 || a == N-1));
\\ Gamma_1(N) via Gamma_0(N): much faster
G1(N) = msfarey(G0(N), g -> my(a=g[1,1]%N); a==1 || a==N-1);
\\ Gamma(N)
G(N) = msfarey(G1(N), g -> g[1,2]%N==0);
G1_0(N) = msfarey(G0(N), x -> x[1,2]%N==0);
G1_0(N1,N2) = msfarey(G0(1), x -> x[2,1]%N1==0 && x[1,2]%N2==0);
\\ Gamma_0(91) has 4 elliptic points of order 3, Gamma_1(91) has none
D0 = mspolygon(G0(91), 2)[4];
D1 = mspolygon(G1(91), 2)[4];
write("F.tex", "\\documentclass{article}\\usepackage{tikz}\\begin{document}", \
D0, "\n", D1, "\\end{document}");
```

msfromcusp(c)

Returns the modular symbol attached to the cusp c, where M is a modular symbol space of level N, attached to $G = \Gamma_0(N)$. The cusp c in $\mathbb{P}^1(\mathbb{Q})/G$ is given either as oo (=(1:0)) or as a rational number a/b (=(a:b)). The attached symbol maps the path $[b] - [a] \in Div^0(\mathbb{P}^1(\mathbb{Q}))$ to $E_c(b) - E_c(a)$, where $E_c(r)$ is 0 when r! = c and $X^{k-2} ||_{\gamma_r}$ otherwise, where $\gamma_r \cdot r = (1:0)$. These symbols span the Eisenstein subspace of M.

```
? M = msinit(2,8); \\ M_8(Gamma_0(2))
? E = mseisenstein(M);
? E[1] \\ two-dimensional
%3 =
[0 - 10]
[0 - 15]
[0 - 3]
[1 \ 0]
? s = msfromcusp(M,oo)
\%4 = [0, 0, 0, 1] \sim
? mseval(M, s)
\%5 = [1, 0]
? s = msfromcusp(M, 1)
\%6 = [-5/16, -15/32, -3/32, 0] \sim
? mseval(M,s)
\%7 = [-x^{6}, -6^{*}x^{5} - 15^{*}x^{4} - 20^{*}x^{3} - 15^{*}x^{2} - 6^{*}x - 1]
```

In case M was initialized with a nonzero *sign*, the symbol is given in terms of the fixed basis of the whole symbol space, not the + or - part (to which it need not belong).

```
? M = msinit(2,8, 1); \\ M_8(Gamma_0(2))^+
? E = mseisenstein(M);
? E[1] \\ still two-dimensional, in a smaller space
```

%3 =
[0 -10]
[0 3]
[-1 0]
? s = msfromcusp(M,oo) \\ in terms of the basis for M_8(Gamma_0(2)) !
%4 = [0, 0, 0, 1]~
? mseval(M, s) \\ same symbol as before
%5 = [1, 0]

msfromell(sign)

Let E/\mathbb{Q} be an elliptic curve of conductor N. For $\varepsilon = 1$, we define the (cuspidal, new) modular symbol x^{ε} in $H^1_c(X_0(N), \mathbb{Q})^{\varepsilon}$ attached to E. For all primes p not dividing N we have $T_p(x^{\varepsilon}) = a_p x^{\varepsilon}$, where $a_p = p + 1 - \#E(\mathbb{F}_p)$.

Let $\Omega^+ = E.omega[1]$ be the real period of E (integration of the Néron differential $dx/(2y + a_1x + a_3)$ on the connected component of $E(\mathbb{R})$, i.e. the generator of $H_1(E, \mathbb{Z})^+$) normalized by $\Omega^+ > 0$. Let $i\Omega^-$ the integral on a generator of $H_1(E, \mathbb{Z})^-$ with $\Omega^- \in \mathbb{R}_{>0}$. If $c_o o$ is the number of connected components of $E(\mathbb{R})$, Ω^- is equal to $(-2/c_o o)ximag(E.omega[2])$. The complex modular symbol is defined by

$$F:\delta \to 2i\pi \int_{\delta} f(z)dz$$

The modular symbols x^{ε} are normalized so that $F = x^{+}\Omega^{+} + x^{-}i\Omega^{-}$. In particular, we have

$$x^{+}([0] - [oo]) = L(E, 1)/\Omega^{+}$$

which defines x unless L(E, 1) = 0. Furthermore, for all fundamental discriminants D such that $\varepsilon D > 0$, we also have

$$\sum_{0 < a < \|D\|} (D\|a) x^{\varepsilon} ([a/\|D\|] - [oo]) = L(E, (D\|.), 1) / \Omega^{\varepsilon},$$

where (D||.) is the Kronecker symbol. The period Ω^- is also $2/c_o ox$ the real period of the twist $E^{(-4)} = elltwist(E, -4)$.

This function returns the pair [M, x], where M is msinit(N, 2) and x is x^{sign} as above when sign = 1, and $x = [x^+, x^-, L_E]$ when sign is 0, where L_E is a matrix giving the canonical \mathbb{Z} -lattice attached to E in the sense of mslattice applied to $\mathbb{Q}x^+ + \mathbb{Q}x^-$. Explicitly, it is generated by (x^+, x^-) when $E(\mathbb{R})$ has two connected components and by $(x^+ - x^-, 2x^-)$ otherwise.

The modular symbols x are given as a t_COL (in terms of the fixed basis of Hom_G(Δ_0, \mathbb{Q}) chosen in M).

```
? E=ellinit([0,-1,1,-10,-20]); \\ X_0(11)
? [M,xp]= msfromell(E,1);
? xp
%3 = [1/5, -1/2, -1/2]~
? [M,x]= msfromell(E);
? x \\ x^+, x^- and L_E
%5 = [[1/5, -1/2, -1/2]~, [0, 1/2, -1/2]~, [1/5, 0; -1, 1; 0, -1]]
? p = 23; (mshecke(M,p) - ellap(E,p))*x[1]
%6 = [0, 0, 0]~ \\ true at all primes, including p = 11; same for x[2]
? (mshecke(M,p) - ellap(E,p))*x[3] == 0
%7 = 1
```

Instead of a single curve E, one may use instead a vector of *isogenous* curves. The function then returns M and the vector of attached modular symbols.

msfromhecke(v, H)

Given a msinit M and a vector v of pairs [p, P] (where p is prime and P is a polynomial with integer coefficients), return a basis of all modular symbols such that $P(T_p)(s) = 0$. If H is present, it must be a Hecke-stable subspace and we restrict to $s \in H$. When T_p has a rational eigenvalue and $P(x) = x - a_p$ has degree 1, we also accept the integer a_p instead of P.

```
? E = ellinit([0,-1,1,-10,-20]) \\11a1
? ellap(E,2)
%2 = -2
? ellap(E,3)
\%3 = -1
? M = msinit(11, 2);
? S = msfromhecke(M, [[2,-2],[3,-1]])
%5 =
[ 1 1]
[-5 0]
[ 0 -5]
? mshecke(M, 2, S)
%6 =
[-2 0]
[ 0 -2]
? M = msinit(23,4);
? S = msfromhecke(M, [[5, x^4-14*x^3-244*x^2+4832*x-19904]]);
? factor( charpoly(mshecke(M,5,S)) )
%9 =
[x^4 - 14^*x^3 - 244^*x^2 + 4832^*x - 19904 2]
```

msgetlevel()

M being a full modular symbol space, as given by msinit, return its level N.

msgetsign()

M being a full modular symbol space, as given by msinit, return its sign: 1 or 0 (unset).

```
? M = msinit(11,4, 1);
? msgetsign(M)
%2 = 1
? M = msinit(11,4);
? msgetsign(M)
%4 = 0
```

msgetweight()

M being a full modular symbol space, as given by msinit, return its weight *k*.

```
? M = msinit(11,4);
? msgetweight(M)
%2 = 4
```

```
mshecke(p, H)
```

M being a full modular symbol space, as given by msinit, p being a prime number, and H being a Hecke-stable subspace (M if omitted) return the matrix of T_p acting on H (U_p if p divides N). Result is undefined if H is not stable by T_p (resp. U_p).

```
? M = msinit(11,2); \\ M_2(Gamma_0(11))
? T2 = mshecke(M, 2)
%2 =
[3 0 0]
[1 - 2 0]
[1 0 -2]
? M = msinit(11,2, 1); \\ M_2(Gamma_0(11))^+
? T2 = mshecke(M, 2)
%4 =
[ 3 0]
[-1 -2]
? N = msnew(M)[1] \\ Q-basis of new cuspidal subspace
%5 =
[-2]
[-5]
? p = 1009; mshecke(M, p, N) \\ action of T_1009 on N
%6 =
[-10]
? ellap(ellinit("11a1"), p)
\%7 = -10
```

msinit(V, sign)

Given G a finite index subgroup of $SL(2,\mathbb{Z})$ and a finite dimensional representation V of $GL(2,\mathbb{Q})$, creates a space of modular symbols, the G-module $\operatorname{Hom}_G(Div^0(\mathbb{P}^1(\mathbb{Q})), V)$. This is canonically isomorphic to $H^1_c(X(G), V)$, and allows to compute modular forms for G. If sign is present and nonzero, it must be 1 and we consider the subspace defined by $Ker(\sigma - sign)$, where σ is induced by [-1,0;0,1]. Currently the only supported groups are the $\Gamma_0(N)$, coded by the integer N > 0. The only supported representation is $V_k = \mathbb{Q}[X, Y]_{k-2}$, coded by the integer $k \ge 2$.

```
? M = msinit(11,2); msdim(M) \\ Gamma0(11), weight 2
%1 = 3
? mshecke(M,2) \\ T_2 acting on M
%2 =
[3 1 1]
[0 -2 0]
[0 0 -2]
? msstar(M) \\ * involution
%3 =
[1 0 0]
```

```
[0 0 1]
[0 1 0]
? Mp = msinit(11,2, 1); msdim(Mp) \\ + part
%4 = 2
? mshecke(Mp,2) \\ T_2 action on M^+
%5 =
[3 2]
[0 -2]
? msstar(Mp)
%6 =
[1 0]
[0 1]
```

msissymbol(s)

M being a full modular symbol space, as given by msinit, check whether s is a modular symbol attached to M. If A is a matrix, check whether its columns represent modular symbols and return a 0 - 1 vector.

```
? M = msinit(7,8, 1); \\ M_8(Gamma_0(7))^+
? A = msnew(M)[1];
? s = A[,1];
? msissymbol(M, s)
\%4 = 1
? msissymbol(M, A)
\%5 = [1, 1, 1]
? S = mseval(M,s);
? msissymbol(M, S)
%7 = 1
? [q,R] = mspathgens(M); q
%8 = [[+oo, 0], [0, 1/2], [1/2, 1]]
? \#R \setminus 3 relations among the generators g_i
\%9 = 3
? T = S; T[3] ++; \\ randomly perturb S(g_3)
? msissymbol(M, T)
\%11 = 0 \setminus  no longer satisfies the relations
```

mslattice(H)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$ and $V_k = \mathbb{Q}[x, y]_{k-2}$. Let M be a full modular symbol space, as given by msinit and let H be a subspace, e.g. as given by mscuspidal. This function returns a canonical \mathbb{Z} structure on H defined as follows. Consider the map $c : M = \operatorname{Hom}_{\Gamma_0(N)}(\Delta_0, V_k) \to H^1(\Gamma_0(N), V_k)$ given by $\phi : -- > class(\gamma \to \phi(0, \gamma^{-1}0))$. Let $L_k = \mathbb{Z}[x, y]_{k-2}$ be the natural \mathbb{Z} -structure of V_k . The result of mslattice is a \mathbb{Z} -basis of the inverse image by c of $H^1(\Gamma_0(N), L_k)$ in the space of modular symbols generated by H.

For user convenience, H can be defined by a matrix representing the \mathbb{Q} -basis of H (in terms of the canonical \mathbb{Q} -basis of M fixed by msinit and used to represent modular symbols).

If omitted, H is the cuspidal part of M as given by mscuspidal. The Eisenstein part $\operatorname{Hom}_{\Gamma_0(N)}(Div(\mathbb{P}^1(\mathbb{Q})), V_k)$ is in the kernel of c, so the result has no meaning for the Eisenstein part H.

```
? M=msinit(11,2);
? [S,E] = mscuspidal(M,1); S[1] \\ a primitive Q-basis of S
%2 =
[ 1 1]
[-5 0]
[ 0 -5]
? mslattice(M,S)
%3 =
[-1/5 - 1/5]
[ 1 0]
[01]
? mslattice(M.E)
%4 =
[1]
[0]
[0]
? M=msinit(5,4);
? S=mscuspidal(M); S[1]
%6 =
[ 7 20]
[ 3 3]
[-10 -23]
[-30 -30]
? mslattice(M,S)
%7 =
[-1/10 -11/130]
[0 - 1/130]
[ 1/10 6/65]
[ 0 1/13]
```

msnew()

M being a full modular symbol space, as given by msinit, return the *new* part of its cuspidal subspace. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

? M = msinit(11,8, 1); \\ M_8(Gamma_0(11))^+ ? N = msnew(M); ? #N[1] \\ 6-dimensional %3 = 6

msomseval(PHI, path)

Return the vectors of moments of the *p*-adic distribution attached to the path path by the overconvergent modular symbol PHI.

```
? M = msinit(3,6,1);
? Mp= mspadicinit(M,5,10);
? phi = [5,-3,-1]~;
? msissymbol(M,phi)
%4 = 1
? PHI = mstooms(Mp,phi);
? ME = msomseval(Mp,PHI,[oo, 0]);
```

mspadicL(s, r)

Returns the value (or r-th derivative) on a character χ^s of \mathbb{Z}_p^* of the p-adic L-function attached to mu.

Let Φ be the *p*-adic distribution-valued overconvergent symbol attached to a modular symbol ϕ for $\Gamma_0(N)$ (eigenvector for $T_N(p)$ for the eigenvalue a_p). Then $L_p(\Phi, \chi^s) = L_p(\mu, s)$ is the *p*-adic *L* function defined by

$$L_p(\Phi,\chi^s) = \int_{\mathbb{Z}_p^*} \chi^s(z) d\mu(z)$$

where μ is the distribution on \mathbb{Z}_p^* defined by the restriction of $\Phi([oo] - [0])$ to \mathbb{Z}_p^* . The *r*-th derivative is taken in direction $\langle \chi \rangle$:

$$L_p^{(r)}(\Phi,\chi^s) = \int_{\mathbb{Z}_p^*} \chi^s(z) (\log z)^r d\mu(z).$$

In the argument list,

- mu is as returned by mspadicmoments (distributions attached to Φ by restriction to discs $a + p^{\nu} \mathbb{Z}_p$, (a, p) = 1).
- s = [s₁, s₂] with s₁ ∈ ℤ ⊂ ℤ_p and s₂modp − 1 or s₂mod2 for p = 2, encoding the p-adic character χ^s := < χ >^{s₁} τ^{s₂}; here χ is the cyclotomic character from Gal(ℚ_p(μ_{p°o})/ℚ_p) to ℤ^{*}_p, and τ is the Teichmüller character (for p > 2 and the character of order 2 on (ℤ/4ℤ)* if p = 2); for convenience, the character [s, s] can also be represented by the integer s.

When a_p is a *p*-adic unit, L_p takes its values in \mathbb{Q}_p . When a_p is not a unit, it takes its values in the two-dimensional \mathbb{Q}_p -vector space $D_{cris}(M(\phi))$ where $M(\phi)$ is the "motive" attached to ϕ , and we return the two *p*-adic components with respect to some fixed \mathbb{Q}_p -basis.

```
? M = msinit(3,6,1); phi=[5, -3, -1]~;
? msissymbol(M,phi)
\%2 = 1
? Mp = mspadicinit(M, 5, 4);
? mu = mspadicmoments(Mp, phi); \\ no twist
\\ End of initializations
? mspadicL(mu,0) \\ L_p(chi^0)
\%5 = 5 + 2*5^2 + 2*5^3 + 2*5^4 + \dots
? mspadicL(mu,1) \\ L_p(chi), zero for parity reasons
\%6 = [0(5^{13})] \sim
? mspadicL(mu,2) \\ L_p(chi^2)
\%7 = 3 + 4*5 + 4*5^2 + 3*5^5 + \dots
? mspadicL(mu,[0,2]) \\ L_p(tau^2)
\%8 = 3 + 5 + 2*5^2 + 2*5^3 + \ldots
? mspadicL(mu, [1,0]) \\ L_p(<chi>)
\%9 = 3*5 + 2*5^2 + 5^3 + 2*5^7 + 5^8 + 5^{10} + 2*5^{11} + 0(5^{13})
? mspadicL(mu,0,1) \\ L_p'(chi^0)
\%10 = 2*5 + 4*5^2 + 3*5^3 + \ldots
? mspadicL(mu, 2, 1) \\ L_p'(chi^2)
%11 = 4*5 + 3*5^2 + 5^3 + 5^4 + \dots
```

Now several quadratic twists: mstooms is indicated.

```
? PHI = mstooms(Mp,phi);
? mu = mspadicmoments(Mp, PHI, 12); \\ twist by 12
? mspadicL(mu)
%14 = 5 + 5^2 + 5^3 + 2*5^4 + ...
```

```
? mu = mspadicmoments(Mp, PHI, 8); \\ twist by 8
? mspadicL(mu)
%16 = 2 + 3*5 + 3*5^2 + 2*5^4 + ...
? mu = mspadicmoments(Mp, PHI, -3); \\ twist by -3 < 0
? mspadicL(mu)
%18 = 0(5^13) \\ always 0, phi is in the + part and D < 0</pre>
```

One can locate interesting symbols of level N and weight k with msnew and mssplit. Note that instead of a symbol, one can input a 1-dimensional Hecke-subspace from mssplit: the function will automatically use the underlying basis vector.

```
? M=msinit(5,4,1); \\ M_4(Gamma_0(5))^+
? L = mssplit(M, msnew(M)); \setminus list of irreducible Hecke-subspaces
? phi = L[1]; \\ one Galois orbit of newforms
? #phi[1] \\... this one is rational
\%4 = 1
? Mp = mspadicinit(M, 3, 4);
? mu = mspadicmoments(Mp, phi);
? mspadicL(mu)
\%7 = 1 + 3 + 3^{3} + 3^{4} + 2^{3}5 + 3^{6} + 0(3^{9})
? M = msinit(11,8, 1); \\ M_8(Gamma_0(11))^+
? Mp = mspadicinit(M, 3, 4);
? L = mssplit(M, msnew(M));
? phi = L[1]; #phi[1] \\ ... this one is two-dimensional
%11 = 2
? mu = mspadicmoments(Mp, phi);
 *** at top-level: mu=mspadicmoments(Mp,ph
 *** /_____
 *** mspadicmoments: incorrect type in mstooms [dim_Q (eigenspace) > 1]
```

mspadicinit(p, n, flag)

M being a full modular symbol space, as given by msinit, and p a prime, initialize technical data needed to compute with overconvergent modular symbols, modulo p^n . If flag is unset, allow all symbols; else initialize only for a restricted range of symbols depending on flag: if flag = 0 restrict to ordinary symbols, else restrict to symbols ϕ such that $T_p(\phi) = a_p \phi$, with $v_p(a_p) >= flag$, which is faster as flag increases. (The fastest initialization is obtained for flag = 0 where we only allow ordinary symbols.) For supersingular eigensymbols, such that $p \parallel a_p$, we must further assume that p does not divide the level.

```
? E = ellinit("11a1");
? [M,phi] = msfromell(E,1);
? ellap(E,3)
%3 = -1
? Mp = mspadicinit(M, 3, 10, 0); \\ commit to ordinary symbols
? PHI = mstooms(Mp,phi);
```

If we restrict the range of allowed symbols with *flag* (for faster initialization), exceptions will occur if $v_p(a_p)$ violates this bound:

```
? E = ellinit("15a1");
? [M,phi] = msfromell(E,1);
? ellap(E,7)
```

```
%3 = 0
? Mp = mspadicinit(M,7,5,0); \\ restrict to ordinary symbols
? PHI = mstooms(Mp,phi)
*** at top-level: PHI=mstooms(Mp,phi)
*** ^_____
*** mstooms: incorrect type in mstooms [v_p(ap) > mspadicinit flag] (t_VEC).
? Mp = mspadicinit(M,7,5); \\ no restriction
? PHI = mstooms(Mp,phi);
```

This function uses $O(N^2(n+k)^2p)$ memory, where N is the level of M.

mspadicmoments(PHI, D)

Given Mp from mspadicinit, an overconvergent eigensymbol PHI from mstooms and a fundamental discriminant D coprime to p, let PHI^D denote the twisted symbol. This function computes the distribution $\mu = PHI^D([0] - oo]) \|\mathbb{Z}_p^*$ restricted to \mathbb{Z}_p^* . More precisely, it returns the moments of the p - 1 distributions $PHI^D([0] - [oo]) \|(a + p\mathbb{Z}_p), 0 < a < p$. We also allow PHI to be given as a classical symbol, which is then lifted to an overconvergent symbol by mstooms; but this is wasteful if more than one twist is later needed.

The returned data μ (*p*-adic distributions attached to PHI) can then be used in mspadicL or mspadicseries. This precomputation allows to quickly compute derivatives of different orders or values at different characters.

? M = msinit(3,6, 1); ? phi = [5,-3,-1]~; ? msissymbol(M, phi) %3 = 1 ? p = 5; mshecke(M,p) * phi \\ eigenvector of T_5, a_5 = 6 %4 = [30, -18, -6]~ ? Mp = mspadicinit(M, p, 10, 0); \\ restrict to ordinary symbols, mod p^10 ? PHI = mstooms(Mp, phi); ? mu = mspadicmoments(Mp, PHI); ? mspadicL(mu) %8 = 5 + 2*5^2 + 2*5^3 + ... ? mu = mspadicmoments(Mp, PHI, 12); \\ twist by 12 ? mspadicL(mu) %10 = 5 + 5^2 + 5^3 + 2*5^4 + ...

mspadicseries(i)

Let Φ be the *p*-adic distribution-valued overconvergent symbol attached to a modular symbol ϕ for $\Gamma_0(N)$ (eigenvector for $T_N(p)$ for the eigenvalue a_p). If μ is the distribution on \mathbb{Z}_p^* defined by the restriction of $\Phi([oo] - [0])$ to \mathbb{Z}_p^* , let

$${}_{p}^{L}(\mu,\tau^{i})(x) = \int_{\mathbb{Z}_{p}^{*}} \tau^{i}(t)(1+x)^{\log_{p}(t)/\log_{p}(u)} d\mu(t)$$

Here, τ is the Teichmüller character and u is a specific multiplicative generator of $1 + 2p\mathbb{Z}_p$. (Namely 1 + p if p > 2 or 5 if p = 2.) To explain the formula, let $G_o o := Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$, let $\chi : G_o o \to \mathbb{Z}_p^*$ be the cyclotomic character (isomorphism) and γ the element of $G_o o$ such that $\chi(\gamma) = u$; then $\chi(\gamma)^{\log_p(t)/\log_p(u)} = \langle t \rangle$.

The *p*-padic precision of individual terms is maximal given the precision of the overconvergent symbol μ .

```
? [M,phi] = msfromell(ellinit("17a1"),1);
? Mp = mspadicinit(M, 5,7);
? mu = mspadicmoments(Mp, phi,1); \\ overconvergent symbol
```

```
? mspadicseries(mu)

%4 = (4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + 4*5^6 + 3*5^7 + 0(5^9)) \setminus (3 + 3*5 + 5^2 + 5^3 + 2*5^4 + 5^6 + 0(5^7))*x \setminus (2 + 3*5 + 5^2 + 4*5^3 + 2*5^4 + 0(5^5))*x^2 \setminus (3 + 4*5 + 4*5^2 + 0(5^3))*x^3 \setminus (3 + 4*5 + 4*5^2 + 0(5^3))*x^3 \setminus (3 + 0(5))*x^4 + 0(x^5)
```

An example with nonzero Teichmüller:

```
? [M,phi] = msfromell(ellinit("11a1"),1);
? Mp = mspadicinit(M, 3,10);
? mu = mspadicmoments(Mp, phi,1);
? mspadicseries(mu, 2)
%4 = (2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + 3^7 + 3^10 + 3^11 + 0(3^12)) \
+ (1 + 3 + 2*3^2 + 3^3 + 3^5 + 2*3^6 + 2*3^8 + 0(3^9))*x \
+ (1 + 2*3 + 3^4 + 2*3^5 + 0(3^6))*x^2 \
+ (3 + 0(3^2))*x^3 + 0(x^4)
```

Supersingular example (not checked)

```
? E = ellinit("17a1"); ellap(E,3)
%1 = 0
? [M,phi] = msfromell(E,1);
? Mp = mspadicinit(M, 3,7);
? mu = mspadicmoments(Mp, phi,1);
? mspadicseries(mu)
%5 = [(2*3^-1 + 1 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 0(3^7)) \
+ (2 + 3^3 + 0(3^5))*x \
+ (1 + 2*3 + 0(3^2))*x^2 + 0(x^3), \
(3^-1 + 1 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 0(3^7)) \
+ (1 + 2*3 + 2*3^2 + 3^3 + 2*3^4 + 0(3^5))*x \
+ (3^-2 + 3^-1 + 0(3^2))*x^2 + 0(3^-2)*x^3 + 0(x^4)]
```

Example with a twist:

```
? E = ellinit("11a1");
? [M,phi] = msfromell(E,1);
? Mp = mspadicinit(M, 3,10);
? mu = mspadicmoments(Mp, phi,5); \\ twist by 5
? L = mspadicseries(mu)
%5 = (2*3^2 + 2*3^4 + 3^5 + 3^6 + 2*3^7 + 2*3^10 + 0(3^12)) \
+ (2*3^2 + 2*3^6 + 3^7 + 3^8 + 0(3^9))*x \
+ (3^3 + 0(3^6))*x^2 + 0(3^2)*x^3 + 0(x^4)
? mspadicL(mu)
%6 = [2*3^2 + 2*3^4 + 3^5 + 3^6 + 2*3^7 + 2*3^10 + 0(3^12)]~
? ellpadicL(E,3,10,,5)
%7 = 2 + 2*3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^6 + 2*3^7 + 0(3^10)
? mspadicseries(mu,1) \\ must be 0
%8 = 0(3^12) + 0(3^9)*x + 0(3^6)*x^2 + 0(3^2)*x^3 + 0(x^4)
```

mspathgens()

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by msinit, return a set of $\mathbb{Z}[G]$ -generators for Δ_0 . The output is [g, R], where g is a minimal system of generators and R the vector of $\mathbb{Z}[G]$ -

relations between the given generators. A relation is coded by a vector of pairs $[a_i, i]$ with $a_i \in \mathbb{Z}[G]$ and i the index of a generator, so that $\sum_i a_i g[i] = 0$.

An element [v] - [u] in Δ_0 is coded by the "path" [u, v], where oo denotes the point at infinity (1 : 0) on the projective line. An element of $\mathbb{Z}[G]$ is either an integer $n (= n[id_2])$ or a "factorization matrix": the first column contains distinct elements g_i of G and the second integers n_i and the matrix codes $\sum n_i[g_i]$:

```
? M = msinit(11,8); \\ M_8(Gamma_0(11))
? [g,R] = mspathgens(M);
? g
%3 = [[+oo, 0], [0, 1/3], [1/3, 1/2]] \\ 3 paths
? #R \\ a single relation
%4 = 1
? r = R[1]; #r \\ ...involving all 3 generators
%5 = 3
? r[1]
%6 = [[1, 1; [1, 1; 0, 1], -1], 1]
? r[2]
%7 = [[1, 1; [7, -2; 11, -3], -1], 2]
? r[3]
%8 = [[1, 1; [8, -3; 11, -4], -1], 3]
```

The given relation is of the form $\sum_i (1 - \gamma_i)g_i = 0$, with $\gamma_i \in \Gamma_0(11)$. There will always be a single relation involving all generators (corresponding to a round trip along all cusps), then relations involving a single generator (corresponding to 2 and 3-torsion elements in the group:

? M = msinit(2,8); \\ M_8(Gamma_0(2))
? [g,R] = mspathgens(M);
? g
%3 = [[+oo, 0], [0, 1]]

Note that the output depends only on the group G, not on the representation V.

mspathlog(p)

Let $\Delta_0 := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by msinit, encoding fixed $\mathbb{Z}[G]$ generators (g_i) of Δ_0 (see mspathgens). A path p = [a, b] between two elements in $\mathbb{P}^1(\mathbb{Q})$ corresponds to $[b] - [a] \in \Delta_0$. The path extremities a and b may be given as t_INT, t_FRAC or oo = (1:0). Finally, we also allow to input a path as a $2x^2$ integer matrix, whose first and second column give a and b respectively, with the convention [x, y] = (x: y) in $\mathbb{P}^1(\mathbb{Q})$.

Returns (p_i) in $\mathbb{Z}[G]$ such that $p = \sum_i p_i g_i$.

```
? M = msinit(2,8); \\ M_8(Gamma_0(2))
? [g,R] = mspathgens(M);
? g
%3 = [[+oo, 0], [0, 1]]
? p = mspathlog(M, [1/2,2/3]);
? p[1]
%5 =
[[1, 0; 2, 1] 1]
? p[2]
%6 =
[[1, 0; 0, 1] 1]
```

```
[[3, -1; 4, -1] 1]
? mspathlog(M, [1,2;2,3]) == p \\ give path via a 2x2 matrix
%7 = 1
```

Note that the output depends only on the group G, not on the representation V.

mspetersson(F, G)

M being a full modular symbol space for $\Gamma = \Gamma_0(N)$, as given by msinit, calculate the intersection product F, G of modular symbols F and G on $M = \text{Hom}_{\Gamma}(\Delta_0, V_k)$ extended to an hermitian bilinear form on $M \otimes \mathbb{C}$ whose radical is the Eisenstein subspace of M.

Suppose that f_1 and f_2 are two parabolic forms. Let F_1 and F_2 be the attached modular symbols

$$F_i(\delta) = \int_{\delta} f_i(z) . (zX + Y)^{k-2} dz$$

and let $F_1^{\mathbb{R}}, F_2^{\mathbb{R}}$ be the attached real modular symbols

$$F_i^{\mathbb{R}}(\delta) = \int_{\delta} \Re(f_i(z).(zX+Y)^{k-2}dz)$$

Then we have

$$= -2 (2i)^{k-2} \cdot \Im (\langle f_1, f_2 \rangle_{Petersson})$$

and

$$F_1, \bar{F}_2 = (2i)^{k-2} < f_1, f_2 >_{Petersson}$$

In weight 2, the intersection product F, G has integer values on the \mathbb{Z} -structure on M given by mslattice and defines a Riemann form on $H^1_{par}(\Gamma, \mathbb{R})$.

For user convenience, we allow F and G to be matrices and return the attached Gram matrix. If F is omitted: treat it as the full modular space attached to M; if G is omitted, take it equal to F.

```
? M = msinit(37,2);
? C = mscuspidal(M)[1];
? mspetersson(M, C)
%3 =
[ 0 -17 -8 -17]
[17 0 -8 -25]
[ 8 8 0 -17]
[17 25 17 0]
? mspetersson(M, mslattice(M,C))
%4 =
[0 -1 \ 0 -1]
[1 \ 0 \ 0 \ -1]
[0 \ 0 \ 0 \ -1]
[1 \ 1 \ 1 \ 0]
? E = ellinit("33a1");
? [M,xpm] = msfromell(E); [xp,xm,L] = xpm;
? mspetersson(M, mslattice(M,L))
%7 =
```

[0]	-3	3]	
[3	0]		
? €	211	mod	degree(E)
%8	=	[3,	-126]

The coefficient 3 in the matrix is the degree of the modular parametrization.

mspolygon(flag)

M describes a subgroup G of finite index in the modular group $PSL_2(\mathbb{Z})$, as given by msinit or a positive integer N (encoding the group $G = \Gamma_0(N)$), or by msfarey (arbitrary subgroup). Return an hyperbolic polygon (Farey symbol) attached to G. More precisely:

- Its vertices are an ordered list in $\mathbb{P}^1(\mathbb{Q})$ and contain a representatives of all cusps.
- Its edges are hyperbolic arcs joining two consecutive vertices; each edge e is labelled by an integer $\mu(e) \in oo, 2, 3$.
- Given a path (a, b) between two elements of P¹(Q), let (a, b) = (b, a) be the opposite path. There is an involution e → e^{*} on the edges. We have μ(e) = oo if and only if e! = e^{*}; when μ(e)! = 3, e is G-equivalent to e^{*}, i.e. there exists γ_e ∈ G such that e = γ_ee^{*}; if μ(e) = 3 there exists γ_e ∈ G of order 3 such that the hyperbolic triangle (e, γ_ee, γ²_ee) is invariant by γ_e. In all cases, to each edge we have attached γ_e ∈ G of order μ(e).

The polygon is given by a triple [E, A, g]

- The list E of its consecutive edges as matrices in $M_2(\mathbb{Z})$.
- The permutation A attached to the involution: if e = E[i] is the *i*-th edge, then A[i] is the index of e^* in E.
- The list g of pairing matrices γ_e . Remark that $\gamma_{e^*} = \gamma_e^{-1}$ if $\mu(e)! = 3$, i.e., $g[i]^{-1} = g[A[i]]$ whenever $i! = A[i] (\mu(g[i]) = 1)$ or $\mu(g[i]) = 2 (g[i]^2 = 1)$. Modulo these trivial relations, the pairing matrices form a system of independent generators of G. Note that γ_e is elliptic if and only if $e^* = e$.

The above data yields a fundamental domain for G acting on Poincaré's half-plane: take the convex hull of the polygon defined by

- The edges in E such that $e! = e^*$ or $e^* = e$, where the pairing matrix γ_e has order 2;
- The edges (r, t) and (t, s) where the edge $e = (r, s) \in E$ is such that $e = e^*$ and γ_e has order 3 and the triangle (r, t, s) is the image of $(0, \exp(2i\pi/3), oo)$ by some element of $PSL_2(\mathbb{Q})$ formed around the edge.

Binary digits of flag mean:

1: return a normalized hyperbolic polygon if set, else a polygon with unimodular edges (matrices of determinant 1). A polygon is normalized in the sense of compact orientable surfaces if the distance $d(a, a^*)$ between an edge a and its image by the involution a^* is less than 2, with equality if and only if a is *linked* with another edge b (a, b, a^* et b^* appear consecutively in E up to cyclic permutation). In particular, the vertices of all edges such that that $d(a, a^*)! = 1$ (distance is 0 or 2) are all equivalent to 0 modulo G. The external vertices of aa^* such that $d(a, a^*) = 1$ are also equivalent to 0; the internal vertices $a \cap a^*$ (a single point), together with 0, form a system of representatives of the cusps of G

 $mathbbP^1(\mathbb{Q})$. This is useful to compute the homology group $H_1(G,\mathbb{Z})$ as it gives a symplectic basis for the intersection pairing. In this case, the number of parabolic matrices (trace 2) in the system of generators G is 2(t-1), where t is the number of non equivalent cusps for G. This is currently only implemented for $G = \Gamma_0(N)$.

2: add graphical representations (in LaTeX form) for the hyperbolic polygon in Poincaré's half-space and the involution $a \rightarrow a^*$ of the Farey symbol. The corresponding character strings can be included in a LaTeX document provided the preamble contains \usepackage{ tikz}.

```
? [V,A,g] = mspolygon(3);
? V
%2 = [[-1, 1; -1, 0], [1, 0; 0, 1], [0, 1; -1, 1]]
? A
%3 = Vecsmall([2, 1, 3])
?g
\%4 = [[-1, -1; 0, -1], [1, -1; 0, 1], [1, -1; 3, -2]]
? [V,A,g, D1,D2] = mspolygon(11,2); \ D1 and D2 contains pictures
? {write("F.tex",
 "\\documentclass{article}\\usepackage{tikz}\\begin{document}"
D1, "\n", D2,
 "\\end{document}");}
? [V1,A1] = mspolygon(6,1); \\ normalized
? V1
\%8 = [[-1, 1; -1, 0], [1, 0; 0, 1], [0, 1; -1, 3],
[1, -2; 3, -5], [-2, 1; -5, 2], [1, -1; 2, -1]]
? A1
%9 = Vecsmall([2, 1, 4, 3, 6, 5])
? [V0,A0] = mspolygon(6); \\ not normalized V[3]^* = V[6], d(V[3],V[6]) = 3
? A0
%11 = Vecsmall([2, 1, 6, 5, 4, 3])
? [V,A] = mspolygon(14, 1);
? A
%13 = Vecsmall([2, 1, 4, 3, 6, 5, 9, 10, 7, 8])
```

One can see from this last example that the (normalized) polygon has the form

 $(a_1, a_1^*, a_2, a_2^*, a_3, a_3^*, a_4, a_5, a_4^*, a_5^*),$

that $X_0(14)$ is of genus 1 (in general the genus is the number of blocks of the form aba^*b^*), has no elliptic points (A has no fixed point) and 4 cusps (number of blocks of the form aa^* plus 1). The vertices of edges a_4 and a_5 all project to 0 in $X_0(14)$: the paths a_4 and a_5 project as loops in $X_0(14)$ and give a symplectic basis of the homology $H_1(X_0(14), \mathbb{Z})$.

```
? [V,A] = mspolygon(15);
? apply(matdet, V) \\ all unimodular
%2 = [1, 1, 1, 1, 1, 1, 1, 1, 1]
? [V,A] = mspolygon(15,1);
? apply(matdet, V) \\ normalized polygon but no longer unimodular edges
%4 = [1, 1, 1, 1, 2, 2, 47, 11, 47, 11]
```

msqexpansion(projH, serprec)

M being a full modular symbol space, as given by msinit, and *projH* being a projector on a Hecke-simple subspace (as given by mssplit), return the Fourier coefficients a_n , $n \le B$ of the corresponding normalized newform. If B is omitted, use seriesprecision.

This function uses a naive $O(B^2d^3)$ algorithm, where d = O(kN) is the dimension of $M_k(\Gamma_0(N))$.

```
? M = msinit(11,2, 1); \\ M_2(Gamma_0(11))^+
? L = mssplit(M, msnew(M));
? msqexpansion(M,L[1], 20)
```

%3 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2] ? ellan(ellinit("11a1"), 20) %4 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]

The shortcut msqexpansion(M, s, B) is available for a symbol *s*, provided it is a Hecke eigenvector:

```
? E = ellinit("11a1");
? [M,S] = msfromell(E); [sp,sm] = S;
? msqexpansion(M,sp,10) \\ in the + eigenspace
%3 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
? msqexpansion(M,sm,10) \\ in the - eigenspace
%4 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
? ellan(E, 10)
%5 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
```

mssplit(H, dimlim)

Let M denote a full modular symbol space, as given by msinit(N, k, 1) or msinit(N, k, -1) and let H be a Hecke-stable subspace of msnew(M) (the full new subspace if H is omitted). This function splits H into Hecke-simple subspaces. If dimlim is present and positive, restrict to subspaces of dimension $\leq = dimlim$. A subspace is given by a structure allowing quick projection and restriction of linear operators; its first component is a matrix with integer coefficients whose columns form a \mathbb{Q} -basis of the subspace.

```
? M = msinit(11,8, 1); \\ M_8(Gamma_0(11))^+
? L = mssplit(M); \\ split msnew(M)
? #L
%3 = 2
? f = msqexpansion(M,L[1],5); f[1].mod
%4 = x^2 + 8*x - 44
? lift(f)
%5 = [1, x, -6*x - 27, -8*x - 84, 20*x - 155]
? g = msqexpansion(M,L[2],5); g[1].mod
%6 = x^4 - 558*x^2 + 140*x + 51744
```

To a Hecke-simple subspace corresponds an orbit of (normalized) newforms, defined over a number field. In the above example, we printed the polynomials defining the said fields, as well as the first 5 Fourier coefficients (at the infinite cusp) of one such form.

msstar(H)

M being a full modular symbol space, as given by msinit, return the matrix of the * involution, induced by complex conjugation, acting on the (stable) subspace H (M if omitted).

? M = msinit(11,2); \\ M_2(Gamma_0(11))
? w = msstar(M);
? w^2 == 1
%3 = 1

mstooms(phi)

Given Mp from mspadicinit, lift the (classical) eigen symbol phi to a *p*-adic distribution-valued overconvergent symbol in the sense of Pollack and Stevens. More precisely, let ϕ belong to the space W of modular symbols of level N, $v_p(N) \leq 1$, and weight k which is an eigenvector for the Hecke operator $T_N(p)$ for a nonzero eigenvalue a_p and let $N_0 = lcm(N, p)$.

Under the action of $T_{N_0}(p)$, ϕ generates a subspace W_{ϕ} of dimension 1 (if p || N) or 2 (if p does not divide N) in the space of modular symbols of level N_0 .

Let $V_p = [p, 0; 0, 1]$ and $C_p = [a_p, p^{k-1}; -1, 0]$. When p does not divide N and a_p is divisible by p, mstooms returns the lift Φ of $(\phi, \phi ||_k V_p)$ such that

$$T_{N_0}(p)\Phi = C_p\Phi$$

When p does not divide N and a_p is not divisible by p, mstooms returns the lift Φ of $\phi - \alpha^{-1} \phi ||_k V_p$ which is an eigenvector of $T_{N_0}(p)$ for the unit eigenvalue where $\alpha^2 - a_p \alpha + p^{k-1} = 0$.

The resulting overconvergent eigensymbol can then be used in mspadicmoments, then mspadicL or mspadicseries.

```
? M = msinit(3,6, 1); p = 5;
? Tp = mshecke(M, p); factor(charpoly(Tp))
%2 =
[x - 3126 2]
[ x - 6 1]
? phi = matker(Tp - 6)[,1] \\ generator of p-Eigenspace, a_p = 6
%3 = [5, -3, -1]~
? Mp = mspadicinit(M, p, 10, 0); \\ restrict to ordinary symbols, mod p^10
? PHI = mstooms(Mp, phi);
? mu = mspadicmoments(Mp, PHI);
? mspadicL(mu)
%7 = 5 + 2*5^2 + 2*5^3 + ...
```

A non ordinary symbol.

```
? M = msinit(4,6,1); p = 3;
? Tp = mshecke(M, p); factor(charpoly(Tp))
%2 =
[x - 244 3]
[x + 12 1]
? phi = matker(Tp + 12)[,1] \setminus a_p = -12 is divisible by p = 3
%3 = [-1/32, -1/4, -1/32, 1]~
? msissymbol(M,phi)
\%4 = 1
? Mp = mspadicinit(M,3,5,0);
? PHI = mstooms(Mp,phi);
*** at top-level: PHI=mstooms(Mp,phi)
 *** mstooms: incorrect type in mstooms [v_p(ap) > mspadicinit flag] (t_VEC).
? Mp = mspadicinit(M,3,5,1);
? PHI = mstooms(Mp,phi);
```

newtonpoly(p)

Gives the vector of the slopes of the Newton polygon of the polynomial x with respect to the prime number p. The n components of the vector are in decreasing order, where n is equal to the degree of x. Vertical slopes occur iff the constant coefficient of x is zero and are denoted by +00.

nextprime()

Finds the smallest pseudoprime (see ispseudoprime) greater than or equal to x. x can be of any real type. Note that if x is a pseudoprime, this function returns x and not the smallest pseudoprime strictly larger than x. To rigorously prove that the result is prime, use isprime.

nfalgtobasis(x)

Given an algebraic number x in the number field nf, transforms it to a column vector on the integral basis : emphasis:`nf.zk`.

? nf = nfinit(y² + 4); ? nf.zk %2 = [1, 1/2*y] ? nfalgtobasis(nf, [1,1]~) %3 = [1, 1]~ ? nfalgtobasis(nf, y) %4 = [0, 2]~ ? nfalgtobasis(nf, Mod(y, y²+4)) %5 = [0, 2]~

This is the inverse function of nfbasistoalg.

nfbasis(dK)

Let T(X) be an irreducible polynomial with integral coefficients. This function returns an integral basis of the number field defined by T, that is a \mathbb{Z} -basis of its maximal order. If present, dK is set to the discriminant of the returned order. The basis elements are given as elements in $K = \mathbb{Q}[X]/(T)$, in Hermite normal form with respect to the \mathbb{Q} -basis $(1, X, ..., X^{\deg T-1})$ of K, lifted to $\mathbb{Q}[X]$. In particular its first element is always 1 and its *i*-th element is a polynomial of degree i - 1 whose leading coefficient is the inverse of an integer: the product of those integers is the index of $\mathbb{Z}[X]/(T)$ in the maximal order \mathbb{Z}_K :

```
? nfbasis(x^2 + 4) \\ Z[X]/(T) has index 2 in Z_K
%1 = [1, x/2]
? nfbasis(x^2 + 4, &D)
%2 = [1, x/2]
? D
%3 = -4
```

This function uses a modified version of the round 4 algorithm, due to David Ford, Sebastian Pauli and Xavier Roblot.

Local basis, orders maximal at certain primes.

Obtaining the maximal order is hard: it requires factoring the discriminant D of T. Obtaining an order which is maximal at a finite explicit set of primes is easy, but it may then be a strict suborder of the maximal order. To specify that we are interested in a given set of places only, we can replace the argument T by an argument [T, listP], where listP encodes the primes we are interested in: it must be a factorization matrix, a vector of integers or a single integer.

- Vector: we assume that it contains distinct *prime* numbers.
- Matrix: we assume that it is a two-column matrix of a (partial) factorization of D; namely the first column contains distinct *primes* and the second one the valuation of D at each of these primes.
- Integer B: this is replaced by the vector of primes up to B. Note that the function will use at least O(B) time: a small value, about 10^5 , should be enough for most applications. Values larger than 2^{32} are not supported.

In all these cases, the primes may or may not divide the discriminant D of T. The function then returns a \mathbb{Z} -basis of an order whose index is not divisible by any of these prime numbers. The result may actually be a global integral basis, in particular if all the prime divisors of the *field* discriminant are included, but this is not guaranteed! Note that nfinit has built-in support for such a check:

```
? K = nfinit([T, listP]);
? nfcertify(K) \\ we computed an actual maximal order
%2 = [];
```

The first line initializes a number field structure incorporating nfbasis([T, listP] in place of a proven integral basis. The second line certifies that the resulting structure is correct. This allows to create an nf structure attached to the number field $K = \mathbb{Q}[X]/(T)$, when the discriminant of T cannot be factored completely, whereas the prime divisors of discK are known. If present, the argument dK is set to the discriminant of the returned order, and is equal to the field discriminant if and only if the order is maximal.

Of course, if *listP* contains a single prime number p, the function returns a local integral basis for $\mathbb{Z}_p[X]/(T)$:

? nfbasis(x^2+x-1001)
%1 = [1, 1/3*x - 1/3]
? nfbasis([x^2+x-1001, [2]])
%2 = [1, x]

The following function computes the index i_T of $\mathbb{Z}[X]/(T)$ in the order generated by the \mathbb{Z} -basis B:

```
nfbasisindex(T, B) = vecprod([denominator(pollead(Q)) | Q <- B]);</pre>
```

In particular, B is a basis of the maximal order if and only if $poldisc(T)/i_T^2$ is equal to the field discriminant. More generally, this formula gives the square of index of the order given by B in \mathbb{Z}_K . For instance, assume that P is a vector of prime numbers containing (at least) all prime divisors of the field discriminant, then the following construct allows to provably compute the field discriminant and to check whether the returned basis is actually a basis of the maximal order

? B = nfbasis([T, P], &D); ? dK = sign(D) * vecprod([p^valuation(D,p) | p<-P]); ? dK * nfbasisindex(T, B)^2 == poldisc(T)

The variable dK contains the field discriminant and the last command returns 1 if and only if B is a \mathbb{Z} -basis of the maximal order. Of course, the nfinit / nfcertify approach is simpler, but it is also more costly.

The Buchmann-Lenstra algorithm.

We now complicate the picture: it is in fact allowed to include *composite* numbers instead of primes in listP (Vector or Matrix case), provided they are pairwise coprime. The result may still be a correct integral basis if the field discriminant factors completely over the actual primes in the list; again, this is not guaranteed. Adding a composite C such that C^2 divides D may help because when we consider C as a prime and run the algorithm, two good things can happen: either we succeed in proving that no prime dividing C can divide the index (without actually needing to find those primes), or the computation exhibits a nontrivial zero divisor, thereby factoring C and we go on with the refined factorization. (Note that including a C such that C^2 does not divide D is useless.) If neither happen, then the computed basis need not generate the maximal order. Here is an example:

```
? B = 10^5;
? listP = factor(poldisc(T), B); \\ primes <= B dividing D + cofactor
? basis = nfbasis([T, listP], &D)
```

If the computed discriminant D factors completely over the primes less than B (together with the primes contained in the addprimes table), then everything is certified: D is the field discriminant and basis generates the maximal order. This can be tested as follows:

```
F = factor(D, B); P = F[,1]; E = F[,2];
for (i = 1, #P,
if (P[i] > B && !isprime(P[i]), warning("nf may be incorrect")));
```

This is a sufficient but not a necessary condition, hence the warning, instead of an error.

The function nfcertify speeds up and automates the above process:

? B = 10^5; ? nf = nfinit([T, B]); ? nfcertify(nf) %3 = [] \\ nf is unconditionally correct ? [basis, disc] = [nf.zk, nf.disc];

nfbasistoalg(*x*)

Given an algebraic number x in the number field nf, transforms it into t_POLMOD form.

```
? nf = nfinit(y<sup>2</sup> + 4);
? nf.zk
%2 = [1, 1/2*y]
? nfbasistoalg(nf, [1,1]~)
%3 = Mod(1/2*y + 1, y<sup>2</sup> + 4)
? nfbasistoalg(nf, y)
%4 = Mod(y, y<sup>2</sup> + 4)
? nfbasistoalg(nf, Mod(y, y<sup>2</sup>+4))
%5 = Mod(y, y<sup>2</sup> + 4)
```

This is the inverse function of nfalgtobasis.

nfcertify()

nf being as output by nfinit, checks whether the integer basis is known unconditionally. This is in particular useful when the argument to nfinit was of the form [T, listP], specifying a finite list of primes when p-maximality had to be proven, or a list of coprime integers to which Buchmann-Lenstra algorithm was to be applied.

The function returns a vector of coprime composite integers. If this vector is empty, then nf.zk and nf.disc are correct. Otherwise, the result is dubious. In order to obtain a certified result, one must completely factor each of the given integers, then addprime each of their prime factors, then check whether nfdisc(nf.pol) is equal to nf.disc.

nfcompositum(*P*, *Q*, *flag*)

Let *nf* be a number field structure attached to the field *K* and let *P* and *Q* be squarefree polynomials in K[X] in the same variable. Outputs the simple factors of the étale *K*-algebra A = K[X,Y]/(P(X),Q(Y)). The factors are given by a list of polynomials *R* in K[X], attached to the number field K[X]/(R), and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. In this case, there will be a single factor R if and only if the number fields defined by P and Q are linearly disjoint (their intersection is K).

The binary digits of flag mean

1: outputs a vector of 4-component vectors [R, a, b, k], where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of K[X]/(R). Finally, k is a small integer such that b + ka = X modulo R.

2: assume that P and Q define number fields that are linearly disjoint: both polynomials are irreducible and the corresponding number fields have no common subfield besides K. This allows to save a costly factorization over K. In this case return the single simple factor instead of a vector with one element.

A compositum is often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is an example involving the field $K(\zeta_5, 5^{1/10}), K = \mathbb{Q}(\sqrt{5})$:

```
? K = nfinit(y^2-5);
? L = nfcompositum(K, x^5 - y, polcyclo(5), 1); \\ list of [R,a,b,k]
? [R, a] = L[1]; \\ pick the single factor, extract R, a (ignore b,k)
? lift(R) \setminus defines the compositum
\%4 = x^{10} + (-5/2*y + 5/2)*x^{9} + (-5*y + 20)*x^{8} + (-20*y + 30)*x^{7} + \langle
(-45/2*y + 145/2)*x^{6} + (-71/2*y + 121/2)*x^{5} + (-20*y + 60)*x^{4} + 
(-25*y + 5)*x^3 + 45*x^2 + (-5*y + 15)*x + (-2*y + 6)
? a^5 - y \setminus a fifth root of y
%5 = 0
? [T, X] = rnfpolredbest(K, R, 1);
? lift(T) \setminus simpler defining polynomial for K[x]/(R)
\%7 = x^{10} + (-11/2*y + 25/2)
? liftall(X) \setminus root of R in K[x]/(T(x))
\%8 = (3/4*y + 7/4)*x^7 + (-1/2*y - 1)*x^5 + 1/2*x^2 + (1/4*y - 1/4)
? a = subst(a.pol, 'x, X); \setminus a in the new coordinates
? liftall(a)
\%10 = (-3/4*y - 7/4)*x^7 - 1/2*x^2
? a^5 - y
\%11 = 0
```

The main variables of P and Q must be the same and have higher priority than that of nf (see varhigher and varlower).

nfdetint(*x*)

Given a pseudo-matrix x, computes a nonzero ideal contained in (i.e. multiple of) the determinant of x. This is particularly useful in conjunction with nfhnfmod.

nfdisc()

field discriminant of the number field defined by the integral, preferably monic, irreducible polynomial T(X). Returns the discriminant of the number field $\mathbb{Q}[X]/(T)$, using the Round 4 algorithm.

Local discriminants, valuations at certain primes.

As in nfbasis, the argument T can be replaced by [T, listP], where listP is as in nfbasis: a vector of pairwise coprime integers (usually distinct primes), a factorization matrix, or a single integer. In that case, the function returns the discriminant of an order whose basis is given by nfbasis(T,listP), which need not be the maximal order, and whose valuation at a prime entry in listP is the same as the valuation of the field discriminant.

In particular, if listP is [p] for a prime p, we can return the p-adic discriminant of the maximal order of $\mathbb{Z}_p[X]/(T)$, as a power of p, as follows:

```
? padicdisc(T,p) = p^valuation(nfdisc([T,[p]]), p);
? nfdisc(x^2 + 6)
%2 = -24
? padicdisc(x^2 + 6, 2)
%3 = 8
? padicdisc(x^2 + 6, 3)
%4 = 3
```

The following function computes the discriminant of the maximal order under the assumption that P is a vector of prime numbers containing (at least) all prime divisors of the field discriminant:

globaldisc(T, P) =
{ my (D = nfdisc([T, P]));
sign(D) * vecprod([p^valuation(D,p) | p <-P]);
}
? globaldisc(x^2 + 6, [2, 3, 5])
%1 = -24</pre>

nfdiscfactors()

Given a polynomial T with integer coefficients, return [D, faD] where D is nfdisc(T) and faD is the factorization of ||D||. All the variants [T,listP] are allowed (see ??nfdisc), in which case faD is the factorization of the discriminant underlying order (which need not be maximal at the primes not specified by listP) and the factorization may contain large composites.

```
? T = x^3 - 6021021*x^2 + 12072210077769*x - 8092423140177664432;
? [D,faD] = nfdiscfactors(T); print(faD); D
[3, 3; 500009, 2]
%2 = -6750243002187]
? T = x^3 + 9*x^2 + 27*x - 125014250689643346789780229390526092263790263725;
? [D,faD] = nfdiscfactors(T); print(faD); D
[3, 3; 1000003, 2]
%4 = -27000162000243
? [D,faD] = nfdiscfactors([T, 10^3]); print(faD)
[3, 3; 125007125141751093502187, 2]
```

In the final example, we only get a partial factorization, which is only guaranteed correct at primes $\langle = 10^3$.

The function also accept number field structures, for instance as output by nfinit, and returns the field discriminant and its factorization:

```
? T = x^3 + 9*x^2 + 27*x - 125014250689643346789780229390526092263790263725;
? nf = nfinit(T); [D,faD] = nfdiscfactors(T); print(faD); D
%2 = -27000162000243
? nf.disc
%3 = -27000162000243
```

nfeltadd(*x*, *y*)

Given two elements x and y in nf, computes their sum x + y in the number field nf.

```
? nf = nfinit(1+x^2);
? nfeltadd(nf, 1, x) \\ 1 + I
%2 = [1, 1]~
```

nfeltdiv(*x*, *y*)

Given two elements x and y in nf, computes their quotient x/y in the number field nf.

nfeltdiveuc(x, y)

Given two elements x and y in nf, computes an algebraic integer q in the number field nf such that the components of x - qy are reasonably small. In fact, this is functionally identical to round(nfdiv(:emphasis:`nf,x,y))`.

nfeltdivmodpr(x, y, pr)

This function is obsolete, use nfmodpr.

Given two elements x and y in nf and pr a prime ideal in modpr format (see nfmodprinit), computes their quotient x/y modulo the prime ideal pr.

nfeltdivrem(x, y)

Given two elements x and y in nf, gives a two-element row vector [q, r] such that x = qy + r, q is an algebraic integer in nf, and the components of r are reasonably small.

nfeltembed(*x*, *pl*, *precision*)

Given an element x in the number field nf, return the (real or) complex embeddings of x specified by optional argument pl, at the current realprecision:

- *pl* omitted: return the vector of embeddings at all $r_1 + r_2$ places;
- *pl* an integer between 1 and r₁ + r₂: return the *i*-th embedding of x, attached to the *i*-th root of nf.pol, i.e. nf.roots:math:`[i]`;
- pl a vector or t_VECSMALL: return the vector of embeddings; the *i*-th entry gives the embedding at the place attached to the pl[*i*]-th real root of nf.pol.

```
? nf = nfinit('y^3 - 2);
? nf.sign
%2 = [1, 1]
? nfeltembed(nf, 'y)
%3 = [1.25992[...], -0.62996[...] + 1.09112[...]*I]]
? nfeltembed(nf, 'y, 1)
%4 = 1.25992[...]
? nfeltembed(nf, 'y, 3) \\ there are only 2 arch. places
*** at top-level: nfeltembed(nf,'y,3)
*** ^------
*** nfeltembed: domain error in nfeltembed: index > 2
```

nfeltmod(x, y)

Given two elements x and y in nf, computes an element r of nf of the form r = x - qy with q and algebraic integer, and such that r is small. This is functionally identical to

x - nfmul(nf, round(nfdiv(nf, x, y)), y).

nfeltmul(x, y)

Given two elements x and y in nf, computes their product x * y in the number field nf.

nfeltmulmodpr(x, y, pr)

This function is obsolete, use nfmodpr.

Given two elements x and y in nf and pr a prime ideal in modpr format (see nfmodprinit), computes their product x * y modulo the prime ideal pr.

nfeltnorm(x)

Returns the absolute norm of x.

nfeltpow(*x*, *k*)

Given an element x in nf, and a positive or negative integer k, computes x^k in the number field nf.

nfeltpowmodpr(*x*, *k*, *pr*)

This function is obsolete, use nfmodpr.

Given an element x in nf, an integer k and a prime ideal pr in modpr format (see nfmodprinit), computes x^k modulo the prime ideal pr.

nfeltreduce(a, id)

Given an ideal *id* in Hermite normal form and an element a of the number field nf, finds an element r in nf such that a - r belongs to the ideal and r is small.

nfeltreducemodpr(x, pr)

This function is obsolete, use nfmodpr.

Given an element x of the number field nf and a prime ideal pr in modpr format compute a canonical representative for the class of x modulo pr.

nfeltsign(x, pl)

Given an element x in the number field nf, returns the signs of the real embeddings of x specified by optional argument pl:

- *pl* omitted: return the vector of signs at all r_1 real places;
- *pl* an integer between 1 and *r*₁: return the sign of the *i*-th embedding of *x*, attached to the *i*-th real root of nf.pol, i.e. nf.roots:math:`[i]`;
- pl a vector or t_VECSMALL: return the vector of signs; the *i*-th entry gives the sign at the real place attached to the pl[i]-th real root of nf.pol.

```
? nf = nfinit(polsubcyclo(11,5,'y)); \\ Q(cos(2 pi/11))
? nf.sign
%2 = [5, 0]
? x = Mod('y, nf.pol);
? nfeltsign(nf, x)
%4 = [-1, -1, -1, 1, 1]
? nfeltsign(nf, x, 1)
%5 = -1
? nfeltsign(nf, x, [1..4])
%6 = [-1, -1, -1, 1]
? nfeltsign(nf, x, 6) \\ there are only 5 real embeddings
*** at top-level: nfeltsign(nf,x,6)
*** ^------
*** nfeltsign: domain error in nfeltsign: index > 5
```

nfelttrace(*x*)

Returns the absolute trace of x.

nfeltval(*x*, *pr*, *y*)

Given an element x in nf and a prime ideal pr in the format output by idealprimedec, computes the valuation v at pr of the element x. The valuation of 0 is +00.

```
? nf = nfinit(x^2 + 1);
? P = idealprimedec(nf, 2)[1];
? nfeltval(nf, x+1, P)
%3 = 1
```

This particular valuation can also be obtained using idealval(:emphasis:`nf,x,:emphasis:pr)`, since x is then converted to a principal ideal.

If the y argument is present, sets $y = x\tau^v$, where τ is a fixed "anti-uniformizer" for pr: its valuation at pr is -1; its valuation is 0 at other prime ideals dividing :emphasis:`pr.p` and nonnegative at all other primes. In other words y is the part of x coprime to pr. If x is an algebraic integer, so is y.

? nfeltval(nf, x+1, P, &y); y %4 = [0, 1]~

For instance if $x = \prod_i x_i^{e_i}$ is known to be coprime to pr, where the x_i are algebraic integers and $e_i \in \mathbb{Z}$ then, if $v_i = nfeltval(nf, x_i, pr, y_i)$, we still have $x = \prod_i y_i^{e_i}$, where the y_i are still algebraic integers but now all of them are coprime to pr. They can then be mapped to the residue field of pr more efficiently than if the product had been expanded beforehand: we can reduce mod pr after each ring operation.

nffactor(T)

Factorization of the univariate polynomial (or rational function) T over the number field nf given by nfinit; T has coefficients in nf (i.e. either scalar, polmod, polynomial or column vector). The factors are sorted by increasing degree.

The main variable of nf must be of *lower* priority than that of T, see priority (in the PARI manual). However if the polynomial defining the number field occurs explicitly in the coefficients of T as modulus of a t_POLMOD or as a t_POL coefficient, its main variable must be *the same* as the main variable of T. For example,

```
? nf = nfinit(y<sup>2</sup> + 1);
? nffactor(nf, x<sup>2</sup> + y); \\ OK
? nffactor(nf, x<sup>2</sup> + Mod(y, y<sup>2</sup>+1)); \\ OK
? nffactor(nf, x<sup>2</sup> + Mod(z, z<sup>2</sup>+1)); \\ WRONG
```

It is possible to input a defining polynomial for *nf* instead, but this is in general less efficient since parts of an nf structure will then be computed internally. This is useful in two situations: when you do not need the nf elsewhere, or when you cannot initialize an nf due to integer factorization difficulties when attempting to compute the field discriminant and maximal order. In all cases, the function runs in polynomial time using Belabas's variant of van Hoeij's algorithm, which copes with hundreds of modular factors.

Caveat. nfinit([T, listP]) allows to compute in polynomial time a conditional *nf* structure, which sets nf. zk to an order which is not guaranteed to be maximal at all primes. Always either use nfcertify first (which may not run in polynomial time) or make sure to input nf.pol instead of the conditional *nf*: nffactor is able to recover in polynomial time in this case, instead of potentially missing a factor.

nffactorback(f, e)

Gives back the nf element corresponding to a factorization. The integer 1 corresponds to the empty factorization.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1s: we return the product of the f[i]. Finally, f can be a regular factorization matrix.

```
? nf = nfinit(y<sup>2</sup>+1);
? nffactorback(nf, [3, y+1, [1,2]~], [1, 2, 3])
%2 = [12, -66]~
? 3 * (I+1)<sup>2</sup> * (1+2*I)<sup>3</sup>
%3 = 12 - 66*I
```

nffactormod(Q, pr)

This routine is obsolete, use nfmodpr and factormod.

Factors the univariate polynomial Q modulo the prime ideal pr in the number field nf. The coefficients of Q belong to the number field (scalar, polmod, polynomial, even column vector) and the main variable of nf must be of lower priority than that of Q (see priority (in the PARI manual)). The prime ideal pr is either in idealprimedec or (preferred) modprimit format. The coefficients of the polynomial factors are lifted to elements of nf:

```
? K = nfinit(y^2+1);
? P = idealprimedec(K, 3)[1];
? nffactormod(K, x^2 + y*x + 18*y+1, P)
%3 =
[x + (2*y + 1) 1]
[x + (2*y + 2) 1]
? P = nfmodprinit(K, P); \\ convert to nfmodprinit format
? nffactormod(K, x^2 + y*x + 18*y+1)
%5 =
[x + (2*y + 1) 1]
[x + (2*y + 2) 1]
```

Same result, of course, here about 10% faster due to the precomputation.

nfgaloisapply(*aut*, *x*)

Let nf be a number field as output by nfinit, and let *aut* be a Galois automorphism of nf expressed by its image on the field generator (such automorphisms can be found using nfgaloisconj). The function computes the action of the automorphism *aut* on the object x in the number field; x can be a number field element, or an ideal (possibly extended). Because of possible confusion with elements and ideals, other vector or matrix arguments are forbidden.

```
? nf = nfinit(x^2+1);
? L = nfgaloisconj(nf)
%2 = [-x, x]~
? aut = L[1]; /* the nontrivial automorphism */
? nfgaloisapply(nf, aut, x)
%4 = Mod(-x, x^2 + 1)
? P = idealprimedec(nf,5); /* prime ideals above 5 */
? nfgaloisapply(nf, aut, P[2]) == P[1]
%6 = 0 \\ !!!!
? idealval(nf, nfgaloisapply(nf, aut, P[2]), P[1])
%7 = 1
```

The surprising failure of the equality test (%7) is due to the fact that although the corresponding prime ideals are equal, their representations are not. (A prime ideal is specified by a uniformizer, and there is no guarantee that applying automorphisms yields the same elements as a direct idealprimedec call.)

The automorphism can also be given as a column vector, representing the image of Mod(x, nf.pol) as an algebraic number. This last representation is more efficient and should be preferred if a given automorphism must be used in many such calls.

```
? nf = nfinit(x<sup>3</sup> - 37*x<sup>2</sup> + 74*x - 37);
? aut = nfgaloisconj(nf)[2]; \\ an automorphism in basistoalg form
%2 = -31/11*x<sup>2</sup> + 1109/11*x - 925/11
? AUT = nfalgtobasis(nf, aut); \\ same in algtobasis form
%3 = [16, -6, 5]~
? v = [1, 2, 3]~; nfgaloisapply(nf, aut, v) == nfgaloisapply(nf, AUT, v)
%4 = 1 \\ same result...
? for (i=1,10<sup>5</sup>, nfgaloisapply(nf, aut, v))
time = 463 ms.
? for (i=1,10<sup>5</sup>, nfgaloisapply(nf, AUT, v))
time = 343 ms. \\ but the latter is faster
```

nfgaloisconj(*flag*, *d*, *precision*)

nf being a number field as output by nfinit, computes the conjugates of a root r of the nonconstant polynomial x = nf[1] expressed as polynomials in r. This also makes sense when the number field is not Galois since some conjugates may lie in the field. nf can simply be a polynomial.

If no flags or flag = 0, use a combination of flag 4 and 1 and the result is always complete. There is no point whatsoever in using the other flags.

If flag = 1, use nfroots: a little slow, but guaranteed to work in polynomial time.

If flag = 4, use galoisinit: very fast, but only applies to (most) Galois fields. If the field is Galois with weakly super-solvable Galois group (see galoisinit), return the complete list of automorphisms, else only the identity element. If present, d is assumed to be a multiple of the least common denominator of the conjugates expressed as polynomial in a root of *pol*.

This routine can only compute \mathbb{Q} -automorphisms, but it may be used to get K-automorphism for any base field K as follows:

```
rnfgaloisconj(nfK, R) = \\ K-automorphisms of L = K[X] / (R)
{
  my(polabs, N,al,S, ala,k, vR);
  R *= Mod(1, nfK.pol); \\ convert coeffs to polmod elts of K
  vR = variable(R);
  al = Mod(variable(nfK.pol),nfK.pol);
  [polabs,ala,k] = rnfequation(nfK, R, 1);
  Rt = if(k==0,R,subst(R,vR,vR-al*k));
  N = nfgaloisconj(polabs) % Rt; \\ Q-automorphisms of L
  S = select(s->subst(Rt, vR, Mod(s,Rt)) == 0, N);
  if (k==0, S, apply(s->subst(s,vR,vR+k*al)-k*al,S));
  }
  K = nfinit(y^2 + 7);
  rnfgaloisconj(K, x^4 - y*x^3 - 3*x^2 + y*x + 1) \\ K-automorphisms of L
```

nfgrunwaldwang(*Lpr*, *Ld*, *pl*, *v*)

Given *nf* a number field in *nf* or *bnf* format, a t_VEC *Lpr* of primes of *nf* and a t_VEC *Ld* of positive integers of the same length, a t_VECSMALL *pl* of length r_1 the number of real places of *nf*, computes a polynomial with coefficients in *nf* defining a cyclic extension of *nf* of minimal degree satisfying certain local conditions:

- at the prime Lpr[i], the extension has local degree a multiple of Ld[i];
- at the *i*-th real place of *nf*, it is complex if pl[i] = -1 (no condition if pl[i] = 0).

The extension has degree the LCM of the local degrees. Currently, the degree is restricted to be a prime power for the search, and to be prime for the construction because of the rnfkummer restrictions.

When nf is \mathbb{Q} , prime integers are accepted instead of prid structures. However, their primality is not checked and the behavior is undefined if you provide a composite number.

Warning. If the number field *nf* does not contain the *n*-th roots of unity where *n* is the degree of the extension to be computed, the function triggers the computation of the *bnf* of $nf(\zeta_n)$, which may be costly.

? nf = nfinit(y^2-5); ? pr = idealprimedec(nf,13)[1]; ? pol = nfgrunwaldwang(nf, [pr], [2], [0,-1], 'x) %3 = x^2 + Mod(3/2*y + 13/2, y^2 - 5)

nfhilbert(a, b, pr)

If pr is omitted, compute the global quadratic Hilbert symbol (a, b) in nf, that is 1 if $x^2 - ay^2 - bz^2$ has a non

trivial solution (x, y, z) in nf, and -1 otherwise. Otherwise compute the local symbol modulo the prime ideal pr, as output by idealprimedec.

nfhnf(x, flag)

Given a pseudo-matrix (A, I), finds a pseudo-basis (B, J) in Hermite normal form of the module it generates. If flag is nonzero, also return the transformation matrix U such that AU = [0||B].

nfhnfmod(*x*, *detx*)

Given a pseudo-matrix (A, I) and an ideal *detx* which is contained in (read integral multiple of) the determinant of (A, I), finds a pseudo-basis in Hermite normal form of the module generated by (A, I). This avoids coefficient explosion. *detx* can be computed using the function nfdetint.

nfinit(flag, precision)

pol being a nonconstant irreducible polynomial in $\mathbb{Q}[X]$, preferably monic and integral, initializes a number field structure (nf) attached to the field K defined by pol. As such, it's a technical object passed as the first argument to most nfxxx functions, but it contains some information which may be directly useful. Access to this information via member functions is preferred since the specific data organization given below may change in the future. Currently, nf is a row vector with 9 components:

nf[1] contains the polynomial pol (:emphasis:`nf.pol`).

nf[2] contains [r1, r2] (:emphasis:`nf.sign`, :emphasis:`nf.r1`, :emphasis:`nf.r2`), the number of real and complex places of K.

nf[3] contains the discriminant d(K) (:emphasis:`nf.disc`) of K.

nf[4] contains the index of nf[1] (:emphasis:`nf.index`), i.e. $[\mathbb{Z}_K : \mathbb{Z}[\theta]]$, where θ is any root of nf[1].

nf[5] is a vector containing 7 matrices M, G, round G, T, MD, TI, MDI and a vector vP defined as follows:

* M is the (r1+r2)xn matrix whose columns represent the numerical values of the conjugates of the elements of the integral basis.

* G is an nxn matrix such that $T2 = {}^t GG$, where T2 is the quadratic form $T_2(x) = \sum ||\sigma(x)||^2$, σ running over the embeddings of K into \mathbb{C} .

* roundG is a rescaled copy of G, rounded to nearest integers.

* T is the nxn matrix whose coefficients are $Tr(\omega_i\omega_j)$ where the ω_i are the elements of the integral basis. Note also that $\det(T)$ is equal to the discriminant of the field K. Also, when understood as an ideal, the matrix T^{-1} generates the codifferent ideal.

* The columns of MD (:emphasis: `nf.diff`) express a \mathbb{Z} -basis of the different of K on the integral basis.

* TI is equal to the primitive part of T^{-1} , which has integral coefficients.

* *MDI* is a two-element representation (for faster ideal product) of d(K) times the codifferent ideal (:emphasis:`nf.disc:math:**nf*.codiff`, which is an integral ideal). This is used in idealinv.

* *vP* is the list of prime divisors of the field discriminant, i.e, the ramified primes (:emphasis:`nf.p`); nfdiscfactors(nf) is the preferred way to access that information.

nf[6] is the vector containing the r1 + r2 roots (:emphasis:`nf.roots`) of nf[1] corresponding to the r1 + r2 embeddings of the number field into \mathbb{C} (the first r1 components are real, the next r2 have positive imaginary part).

nf[7] is a \mathbb{Z} -basis for $d\mathbb{Z}_K$, where $d = [\mathbb{Z}_K : \mathbb{Z}(\theta)]$, expressed on the powers of θ . The multiplication by d ensures that all polynomials have integral coefficients and nf[7]/d (:emphasis:`nf.zk`) is an integral basis for \mathbb{Z}_K . Its first element is guaranteed to be 1. This basis is LLL-reduced with respect to T_2 (strictly speaking, it is a permutation of such a basis, due to the condition that the first element be 1).

nf[8] is the nxn integral matrix expressing the power basis in terms of the integral basis, and finally

nf[9] is the nxn^2 matrix giving the multiplication table of the integral basis.

If a non monic or non integral polynomial is input, nfinit will transform it, and return a structure attached to the new (monic integral) polynomial together with the attached change of variables, see flag = 3. It is allowed, though not very useful given the existence of nfnewprec, to input a *nf* or a *bnf* instead of a polynomial. It is also allowed to input a *rnf*, in which case an nf structure attached to the absolute defining polynomial polabs is returned (*flag* is then ignored).

```
? nf = nfinit(x^3 - 12); \\ initialize number field Q[X] / (X^3 - 12)
? nf.pol \\ defining polynomial
\%2 = x^{3} - 12
? nf.disc \\ field discriminant
\%3 = -972
? nf.index \\ index of power basis order in maximal order
\%4 = 2
? nf.zk \setminus integer basis, lifted to Q[X]
\%5 = [1, x, 1/2*x^2]
? nf.sign \\ signature
\%6 = [1, 1]
? factor(abs(nf.disc )) \\ determines ramified primes
%7 =
[2 2]
[3 5]
? idealfactor(nf, 2)
%8 =
[[2, [0, 0, -1]~, 3, 1, [0, 1, 0]~] 3] \setminus p_2^3
```

Huge discriminants, helping nfdisc.

In case *pol* has a huge discriminant which is difficult to factor, it is hard to compute from scratch the maximal order. The following special input formats are also accepted:

- [pol, B] where pol is a monic integral polynomial and B is the lift of an integer basis, as would be computed by nfbasis: a vector of polynomials with first element 1 (implicitly modulo pol). This is useful if the maximal order is known in advance.
- [pol, B, P] where pol and B are as above (a monic integral polynomial and the lift of an integer basis), and P is the list of ramified primes in the extension.
- [pol, listP] where pol is a rational polynomial and listP specifies a list of primes as in nfbasis. Instead of the maximal order, nfinit then computes an order which is maximal at these particular primes as well as the primes contained in the private prime table, see addprimes. The result has a good chance of being correct when the discriminant nf.disc factors completely over this set of primes but this is not guaranteed. The function nfcertify automates this:

```
? pol = polcompositum(x^5 - 101, polcyclo(7))[1];
? nf = nfinit( [pol, 10^3] );
? nfcertify(nf)
%3 = []
```

A priori, nf.zk defines an order which is only known to be maximal at all primes $\langle = 10^3$ (no prime $\langle = 10^3$ divides nf.index). The certification step proves the correctness of the computation. Had it failed, that particular nf structure could not have been trusted and may have caused routines using it to fail randomly. One particular function that remains trustworthy in all cases is idealprimedec when applied to a prime included in the above list of primes or, more generally, a prime not dividing any entry in nfcertify output.

If flag = 2: pol is changed into another polynomial P defining the same number field, which is as simple as can easily be found using the polredbest algorithm, and all the subsequent computations are done using this new

polynomial. In particular, the first component of the result is the modified polynomial.

If flag = 3, apply polredbest as in case 2, but outputs [nf, Mod(a, P)], where nf is as before and Mod(a, P) = Mod(x, pol) gives the change of variables. This is implicit when pol is not monic or not integral: first a linear change of variables is performed, to get a monic integral polynomial, then polredbest.

nfisideal(*x*)

Returns 1 if x is an ideal in the number field nf, 0 otherwise.

nfisincl(g, flag)

Let f and g define number fields, where f and g are irreducible polynomials in $\mathbb{Q}[X]$ and nf structures as output by nfinit. Tests whether the number field f is conjugate to a subfield of the field g. If they are not, the output is the integer 0. If they are, the output is a vector of polynomials (flag = 0, default) or a single polynomial flag = 1, each polynomial a representing an embedding i.e. being such that g || foa. If either f or g is not irreducible, the result is undefined.

```
? T = x^{6} + 3^{*}x^{4} - 6^{*}x^{3} + 3^{*}x^{2} + 18^{*}x + 10;
U = x^3 + 3^*x^2 + 3^*x - 2
? v = nfisincl(U, T);
\%2 = [24/179*x^{5}-27/179*x^{4}+80/179*x^{3}-234/179*x^{2}+380/179*x+94/179]
? subst(U, x, Mod(v[1],T))
\%3 = Mod(0, x^6 + 3*x^4 - 6*x^3 + 3*x^2 + 18*x + 10)
? #nfisincl(x^2+1, T) \\ two embeddings
\%4 = 2
\\ same result with nf structures
? nfisincl(U, L = nfinit(T)) == v
\%5 = 1
? nfisincl(K = nfinit(U), T) == v
\%6 = 1
? nfisincl(K, L) == v
\%7 = 1
\\ comparative bench: an nf is a little faster, esp. for the subfield
? B = 10^{3}:
? for (i=1, B, nfisincl(U,T))
time = 712 \text{ ms.}
? for (i=1, B, nfisincl(K,T))
time = 485 \text{ ms.}
? for (i=1, B, nfisincl(U,L))
time = 704 \text{ ms.}
? for (i=1, B, nfisincl(K,L))
time = 465 \text{ ms}.
```

Using an *nf* structure for the potential subfield is faster if the structure is already available. On the other hand, the gain in nfisincl is usually not sufficient to make it worthwhile to initialize only for that purpose.

```
? for (i=1, B, nfinit(U))
time = 308 ms.
```

nfisisom(g)

As nfisincl, but tests for isomorphism. More efficient if f or g is a number field structure.

? $f = x^{6} + 30^{*}x^{5} + 495^{*}x^{4} + 1870^{*}x^{3} + 16317^{*}x^{2} - 22560^{*}x + 59648;$? $q = x^6 + 42^*x^5 + 999^*x^4 + 8966^*x^3 + 36117^*x^2 + 21768^*x + 159332;$? $h = x^6 + 30^*x^5 + 351^*x^4 + 2240^*x^3 + 10311^*x^2 + 35466^*x + 58321;$? #nfisisom(f,g) \\ two isomorphisms %3 = 2? nfisisom(f,h) \\ not isomorphic %4 = 0\\ comparative bench ? $K = nfinit(f); L = nfinit(g); B = 10^3;$? for (i=1, B, nfisisom(f,g)) time = 6,124 ms.? for (i=1, B, nfisisom(K,g)) time = 3,356 ms. ? for (i=1, B, nfisisom(f,L)) time = 3,204 ms.? for (i=1, B, nfisisom(K,L)) time = 3,173 ms.

The function is usually very fast when the fields are nonisomorphic, whenever the fields can be distinguished via a simple invariant such as degree, signature or discriminant. It may be slower when the fields share all invariants, but still faster than computing actual isomorphisms:

```
\\ usually very fast when the answer is 'no':
? for (i=1, B, nfisisom(f,h))
time = 32 \text{ ms.}
\\ but not always
? u = x^{6} + 12^{*}x^{5} + 6^{*}x^{4} - 377^{*}x^{3} - 714^{*}x^{2} + 5304^{*}x + 15379
? v = x^{6} + 12^{*}x^{5} + 60^{*}x^{4} + 166^{*}x^{3} + 708^{*}x^{2} + 6600^{*}x + 23353
? nfisisom(u,v)
\%13 = 0
? polsturm(u) == polsturm(v)
\%14 = 1
? nfdisc(u) == nfdisc(v)
%15 = 1
? for(i=1,B, nfisisom(u,v))
time = 1,821 ms.
? K = nfinit(u); L = nfinit(v);
? for(i=1,B, nfisisom(K,v))
time = 232 \text{ ms.}
```

nfislocalpower(pr, a, n)

Let *nf* be a *nf* structure attached to a number field K, let $a \in K$ and let *pr* be a *prid* structure attached to a maximal ideal v. Return 1 if a is an n-th power in the completed local field K_v , and 0 otherwise.

```
? K = nfinit(y^2+1);
? P = idealprimedec(K,2)[1]; \\ the ramified prime above 2
? nfislocalpower(K,P,-1, 2) \\ -1 is a square
%3 = 1
```

```
? nfislocalpower(K,P,-1, 4) \\ ... but not a 4-th power
%4 = 0
? nfislocalpower(K,P,2, 2) \\ 2 is not a square
%5 = 0
? Q = idealprimedec(K,5)[1]; \\ a prime above 5
? nfislocalpower(K,Q, [0, 32]~, 30) \\ 32*I is locally a 30-th power
%7 = 1
```

nfkermodpr(*x*, *pr*)

This function is obsolete, use nfmodpr.

Kernel of the matrix a in \mathbb{Z}_K/pr , where pr is in **modpr** format (see nfmodprinit).

nfmodpr(x, pr)

Map x to a t_FFELT in the residue field modulo pr. The argument pr is either a maximal ideal in idealprimedec format or, preferably, a *modpr* structure from nfmodprinit. The function nfmodprlift allows to lift back to \mathbb{Z}_K .

Note that the function applies to number field elements and not to vector / matrices / polynomials of such. Use apply to convert recursive structures.

```
? K = nfinit(y^3-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K, P, 't);
? K.zk
%4 = [1, 1/5*y, 1/25*y^2]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, t, 2*t + 1]
? %[1].mod
%6 = t^2 + 3*t + 4
? K.index
%7 = 125
```

For clarity, we represent elements in the residue field $\mathbb{F}_5[t]/(T)$ as polynomials in the variable t. Whenever the underlying rational prime does not divide K.index, it is actually the case that t is the reduction of y in $\mathbb{Q}[y]/(K.pol)$ modulo an irreducible factor of K.pol over \mathbb{F}_p . In the above example, 5 divides the index and t is actually the reduction of y/5.

nfmodprinit(pr, v)

Transforms the prime ideal pr into modpr format necessary for all operations modulo pr in the number field nf. The functions nfmodpr and nfmodprlift allow to project to and lift from the residue field. The variable v is used to display finite field elements (see ffgen).

```
? K = nfinit(y^3-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K, P, 't);
? K.zk
%4 = [1, 1/5*y, 1/25*y^2]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, t, 2*t + 1]
? %[1].mod
%6 = t^2 + 3*t + 4
```

(continued	from	previous	nage)
<u>۱</u>	commucu	monn	previous	pu ₅ c)

? K.index		
%7 = 125		

For clarity, we represent elements in the residue field $\mathbb{F}_5[t]/(T)$ as polynomials in the variable t. Whenever the underlying rational prime does not divide K.index, it is actually the case that t is the reduction of y in $\mathbb{Q}[y]/(K.pol)$ modulo an irreducible factor of K.pol over \mathbb{F}_p . In the above example, 5 divides the index and t is actually the reduction of y/5.

nfmodprlift(*x*, *pr*)

Lift the t_FFELT x (from nfmodpr) in the residue field modulo pr to the ring of integers. Vectors and matrices are also supported. For polynomials, use apply and the present function.

The argument pr is either a maximal ideal in idealprimedec format or, preferably, a *modpr* structure from nfmodprinit. There are no compatibility checks to try and decide whether x is attached the same residue field as defined by pr: the result is undefined if not.

The function nfmodpr allows to reduce to the residue field.

```
? K = nfinit(y<sup>3</sup>-250);
? P = idealprimedec(K, 5)[2];
? modP = nfmodprinit(K,P);
? K.zk
%4 = [1, 1/5*y, 1/25*y<sup>2</sup>]
? apply(t->nfmodpr(K,t,modP), K.zk)
%5 = [1, y, 2*y + 1]
? nfmodprlift(K, %, modP)
%6 = [1, 1/5*y, 2/5*y + 1]
? nfeltval(K, %[3] - K.zk[3], P)
%7 = 1
```

nfnewprec(precision)

Transforms the number field nf into the corresponding data using current (usually larger) precision. This function works as expected if nf is in fact a bnf or a bnr (update structure to current precision). If the original bnf structure was *not* computed by bnfinit(,1), then this may be quite slow and even fail: many generators of principal ideals have to be computed and the algorithm may fail because the accuracy is not sufficient to bootstrap the required generators and fundamental units.

nfpolsturm(*T*, *pl*)

Given a polynomial T with coefficients in the number field nf, returns the number of real roots of the s(T) where s runs through the real embeddings of the field specified by optional argument pl:

- pl omitted: all r_1 real places;
- *pl* an integer between 1 and *r*₁: the embedding attached to the *i*-th real root of nf.pol, i.e. nf. roots:math:`[i]`;
- pl a vector or t_VECSMALL: the embeddings attached to the pl[i]-th real roots of nf.pol.

```
? nf = nfinit('y^2 - 2);
? nf.sign
%2 = [2, 0]
? nf.roots
%3 = [-1.414..., 1.414...]
? T = x^2 + 'y;
? nfpolsturm(nf, T, 1) \\ subst(T,y,sqrt(2)) has two real roots
```

```
%5 = 2
? nfpolsturm(nf, T, 2) \\ subst(T,y,-sqrt(2)) has no real root
%6 = 0
? nfpolsturm(nf, T) \\ all embeddings together
%7 = [2, 0]
? nfpolsturm(nf, T, [2,1]) \\ second then first embedding
%8 = [0, 2]
? nfpolsturm(nf, x^3) \\ number of distinct roots !
%9 = [1, 1]
? nfpolsturm(nf, x, 6) \\ there are only 2 real embeddings !
*** at top-level: nfpolsturm(nf,x,6)
*** ^------
*** nfpolsturm: domain error in nfpolsturm: index > 2
```

nfroots(*x*)

Roots of the polynomial x in the number field nf given by nfinit without multiplicity (in \mathbb{Q} if nf is omitted). x has coefficients in the number field (scalar, polmod, polynomial, column vector). The main variable of nf must be of lower priority than that of x (see priority (in the PARI manual)). However if the coefficients of the number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see nffactor).

It is possible to input a defining polynomial for *nf* instead, but this is in general less efficient since parts of an nf structure will then be computed internally. This is useful in two situations: when you do not need the nf elsewhere, or when you cannot initialize an nf due to integer factorization difficulties when attempting to compute the field discriminant and maximal order.

Caveat. nfinit([T, listP]) allows to compute in polynomial time a conditional *nf* structure, which sets nf. zk to an order which is not guaranteed to be maximal at all primes. Always either use nfcertify first (which may not run in polynomial time) or make sure to input nf.pol instead of the conditional *nf*: nfroots is able to recover in polynomial time in this case, instead of potentially missing a factor.

nfrootsof1()

Returns a two-component vector [w, z] where w is the number of roots of unity in the number field nf, and z is a primitive w-th root of unity. It is possible to input a defining polynomial for nf instead.

```
? K = nfinit(polcyclo(11));
? nfrootsof1(K)
%2 = [22, [0, 0, 0, 0, 0, -1, 0, 0, 0, 0]~]
? z = nfbasistoalg(K, %[2]) \\ in algebraic form
%3 = Mod(-x^5, x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1)
? [lift(z^11), lift(z^2)] \\ proves that the order of z is 22
%4 = [-1, -x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 - x^2 - x - 1]
```

This function guesses the number w as the gcd of the $\#k(v)^*$ for unramified v above odd primes, then computes the roots in nf of the w-th cyclotomic polynomial. The algorithm is polynomial time with respect to the field degree and the bitsize of the multiplication table in nf (both of them polynomially bounded in terms of the size of the discriminant). Fields of degree up to 100 or so should require less than one minute.

nfsnf(x, flag)

Given a torsion \mathbb{Z}_K -module x attached to the square integral invertible pseudo-matrix (A, I, J), returns an ideal list $D = [d_1, ..., d_n]$ which is the Smith normal form of x. In other words, x is isomorphic to $\mathbb{Z}_K/d_1 \oplus ... \oplus \mathbb{Z}_K/d_n$ and d_i divides d_{i-1} for $i \ge 2$. If flag is nonzero return [D, U, V], where UAV is the identity.

See ZKmodules (in the PARI manual) for the definition of integral pseudo-matrix; briefly, it is input as a 3component row vector [A, I, J] where $I = [b_1, ..., b_n]$ and $J = [a_1, ..., a_n]$ are two ideal lists, and A is a square
nxn matrix with columns $(A_1, ..., A_n)$, seen as elements in K^n (with canonical basis $(e_1, ..., e_n)$). This data defines the \mathbb{Z}_K module x given by

$$(b_1e_1\oplus\ldots\oplus b_ne_n)/(a_1A_1\oplus\ldots\oplus a_nA_n),$$

The integrality condition is $a_{i,j} \in b_i a_j^{-1}$ for all i, j. If it is not satisfied, then the d_i will not be integral. Note that every finitely generated torsion module is isomorphic to a module of this form and even with $b_i = Z_K$ for all i.

nfsolvemodpr(*a*, *b*, *P*)

This function is obsolete, use nfmodpr.

Let P be a prime ideal in **modpr** format (see nfmodprinit), let a be a matrix, invertible over the residue field, and let b be a column vector or matrix. This function returns a solution of $a \cdot x = b$; the coefficients of x are lifted to *nf* elements.

```
? K = nfinit(y<sup>2</sup>+1);
? P = idealprimedec(K, 3)[1];
? P = nfmodprinit(K, P);
? a = [y+1, y; y, 0]; b = [1, y]~
? nfsolvemodpr(K, a,b, P)
%5 = [1, 2]~
```

nfsplitting(d)

Defining polynomial over \mathbb{Q} for the splitting field of $P \in \mathbb{Q}[x]$, that is the smallest field over which P is totally split. If irreducible, the polynomial P can also be given by a **nf** structure, which is more efficient. If d is given, it must be a multiple of the splitting field degree. Note that if P is reducible the splitting field degree can be smaller than the degree of P.

```
? K = nfinit(x^3-2);
? nfsplitting(K)
%2 = x^6 + 108
? nfsplitting(x^8-2)
%3 = x^16 + 272*x^8 + 64
? S = nfsplitting(x^6-8) // reducible
%4 = x^4+2*x^2+4
? lift(nfroots(subst(S,x,a),x^6-8))
%5 = [-a,a,-1/2*a^3-a,-1/2*a^3,1/2*a^3+a]
```

Specifying the degree of the splitting field can make the computation faster.

```
? nfsplitting(x^17-123);
time = 3,607 ms.
? poldegree(%)
%2 = 272
? nfsplitting(x^17-123,272);
time = 150 ms.
? nfsplitting(x^17-123,273);
*** nfsplitting: Warning: ignoring incorrect degree bound 273
time = 3,611 ms.
```

The complexity of the algorithm is polynomial in the degree d of the splitting field and the bitsize of T; if d is large the result will likely be unusable, e.g. nfinit will not be an option:

```
? nfsplitting(x^6-x-1)
[... degree 720 polynomial deleted ...]
time = 11,020 ms.
```

nfsubfields(*d*, *fl*)

Finds all subfields of degree d of the number field defined by the (monic, integral) polynomial *pol* (all subfields if d is null or omitted). The result is a vector of subfields, each being given by [g, h] (default) or simply g (*flag* = 1), where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf. This routine uses

- Allombert's galoissubfields when nf is Galois (with weakly supersolvable Galois group).
- Klüners's or van Hoeij-Klüners-Novocin algorithm in the general case. The latter runs in polynomial time and is generally superior unless there exists a small unramified prime p such that pol has few irreducible factors modulo p.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, expressed as a famat of polynomials with coefficients in the variable of nf, in which case the van Hoeij-Klüners-Novocin algorithm is used.

```
? pol = x<sup>4</sup> - x<sup>3</sup> - x<sup>2</sup> + x + 1;
? nfsubfields(pol)
%2 = [[x, 0], [x<sup>2</sup> - x + 1, x<sup>3</sup> - x<sup>2</sup> + 1], [x<sup>4</sup> - x<sup>3</sup> - x<sup>2</sup> + x + 1, x]]
? nfsubfields(pol, 1)
%2 = [x, x<sup>2</sup> - x + 1, x<sup>4</sup> - x<sup>3</sup> - x<sup>2</sup> + x + 1]
? y=varhigher("y"); fa = nffactor(pol,subst(pol,x,y));
? #nfsubfields([pol,fa])
%5 = 3
```

nfsubfieldscm(fl)

Compute the maximal CM subfield of *nf*. Return 0 if *nf* does not have a CM subfield, otherwise return [g, h] (default) or g (flag = 1) where g is an absolute equation and h expresses a root of g in terms of the generator of *nf*. Moreover, the CM involution is given by Xmodg(X) : -- > -Xmodg(X), i.e. Xmodg(X) is a totally imaginary element.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, and *nf* is also allowed to be a monic defining polynomial for the number field.

```
? nf = nfinit(x^8 + 20*x^6 + 10*x^4 - 4*x^2 + 9);
? nfsubfieldscm(nf)
%2 = [x^4 + 4480*x^2 + 3612672, 3*x^5 + 58*x^3 + 5*x]
? pol = y^16-8*y^14+29*y^12-60*y^10+74*y^8-48*y^6+8*y^4+4*y^2+1;
? fa = nffactor(pol, subst(pol,y,x));
? nfsubfieldscm([pol,fa])
%5 = [y^8 + ..., ...]
```

nfsubfieldsmax(fl)

Compute the list of maximal subfields of *nf*. The result is a vector as in nfsubfields.

An input of the form [nf, fa] is also allowed, where fa is the factorisation of *nf.pol* over *nf*, and *nf* is also allowed to be a monic defining polynomial for the number field.

norm()

Algebraic norm of x, i.e. the product of x with its conjugate (no square roots are taken), or conjugates for polmods. For vectors and matrices, the norm is taken componentwise and hence is not the L^2 -norm (see norm12). Note that the norm of an element of \mathbb{R} is its square, so as to be compatible with the complex norm.

norml2()

Square of the L^2 -norm of x. More precisely, if x is a scalar, $norml_2(x)$ is defined to be the square of the complex modulus of x (real t_QUAD s are not supported). If x is a polynomial, a (row or column) vector or a matrix, norml2(:math:`x)` is defined recursively as $\sum_i norml_2(x_i)$, where (x_i) run through the components

of x. In particular, this yields the usual $\sum ||x_i||^2$ (resp. $\sum ||x_{i,j}||^2$) if x is a polynomial or vector (resp. matrix) with complex components.

```
? norml2( [ 1, 2, 3 ] ) \\ vector
%1 = 14
? norml2( [ 1, 2; 3, 4] ) \\ matrix
%2 = 30
? norml2( 2*I + x )
%3 = 5
? norml2( [ [1,2], [3,4], 5, 6 ] ) \\ recursively defined
%4 = 91
```

normlp(p, precision)

 L^p -norm of x; sup norm if p is omitted or +oo. More precisely, if x is a scalar, normlp(x, p) is defined to be abs(x). If x is a polynomial, a (row or column) vector or a matrix:

- if p is omitted or +oo, then normlp(:math:`x)` is defined recursively as max_i normlp(x_i)), where (x_i) run through the components of x. In particular, this yields the usual sup norm if x is a polynomial or vector with complex components.
- otherwise, normlp(:math:`x, p)` is defined recursively as $(\sum_i normlp^p(x_i, p))^{1/p}$. In particular, this yields the usual $(\sum ||x_i||^p)^{1/p}$ if x is a polynomial or vector with complex components.

```
? v = [1, -2, 3]; normlp(v) \\ vector
\%1 = 3
? normlp(v, +oo) \setminus same, more explicit
%2 = 3
? M = [1, -2; -3, 4]; normlp(M) \\ matrix
\%3 = 4
? T = (1+I) + I*x^2; normlp(T)
%4 = 1.4142135623730950488016887242096980786
? normlp([[1,2], [3,4], 5, 6]) \\ recursively defined
%5 = 6
? normlp(v, 1)
\%6 = 6
? normlp(M, 1)
\%7 = 10
? normlp(T, 1)
%8 = 2.4142135623730950488016887242096980786
```

numbpart()

Gives the number of unrestricted partitions of n, usually called p(n) in the literature; in other words the number of nonnegative integer solutions to a + 2b + 3c + ... = n. n must be of type integer and $n < 10^{15}$ (with trivial values p(n) = 0 for n < 0 and p(0) = 1). The algorithm uses the Hardy-Ramanujan-Rademacher formula. To explicitly enumerate them, see partitions.

numdiv()

Number of divisors of ||x||. x must be of type integer.

numerator(D)

Numerator of f. This is defined as f * denominator(f, D), see denominator for details. The optional argument D allows to control over which ring we compute the denominator:

- 1: we only consider the underlying Q-structure and the denominator is a (positive) rational integer
- a simple variable, say 'x: all entries as rational functions in K(x) and the denominator is a polynomial in x.

? f = x + 1/y + 1/2; ? numerator(f) \\ a t_POL in x %2 = x + ((y + 2)/(2*y)) ? numerator(f, 1) \\ Q-denominator is 2 %3 = x + ((y + 2)/y) ? numerator(f, y) \\ as a rational function in y %5 = 2*y*x + (y + 2)

omega()

Number of distinct prime divisors of ||x||. x must be of type integer.

? factor(392)
%1 =
[2 3]
[7 2]
? omega(392)
%2 = 2; \\ without multiplicity
? bigomega(392)
%3 = 5; \\ = 3+2, with multiplicity

padicappr(a)

Vector of *p*-adic roots of the polynomial *pol* congruent to the *p*-adic number *a* modulo *p*, and with the same *p*-adic precision as *a*. The number *a* can be an ordinary *p*-adic number (type t_PADIC, i.e. an element of \mathbb{Z}_p) or can be an integral element of a finite *unramified* extension $\mathbb{Q}_p[X]/(T)$ of \mathbb{Q}_p , given as a t_POLMOD Mod(A, T) at least one of whose coefficients is a t_PADIC and *T* irreducible modulo *p*. In this case, the result is the vector of roots belonging to the same extension of \mathbb{Q}_p as *a*. The polynomial *pol* should have exact coefficients; if not, its coefficients are first rounded to \mathbb{Q} or $\mathbb{Q}[X]/(T)$ and this is the polynomial whose roots we consider.

padicfields(N, flag)

Returns a vector of polynomials generating all the extensions of degree N of the field \mathbb{Q}_p of p-adic rational numbers; N is allowed to be a 2-component vector [n, d], in which case we return the extensions of degree n and discriminant p^d .

The list is minimal in the sense that two different polynomials generate nonisomorphic extensions; in particular, the number of polynomials is the number of classes of nonisomorphic extensions. If P is a polynomial in this list, α is any root of P and $K = \mathbb{Q}_p(\alpha)$, then α is the sum of a uniformizer and a (lift of a) generator of the residue field of K; in particular, the powers of α generate the ring of p-adic integers of K.

If flag = 1, replace each polynomial P by a vector [P, e, f, d, c] where e is the ramification index, f the residual degree, d the valuation of the discriminant, and c the number of conjugate fields. If flag = 2, only return the *number* of extensions in a fixed algebraic closure (Krasner's formula), which is much faster.

padicprec(p)

Returns the absolute *p*-adic precision of the object x; this is the minimum precision of the components of x. The result is +00 if x is an exact object (as a *p*-adic):

? padicprec((1 + 0(2⁵)) * x + (2 + 0(2⁴)), 2) %1 = 4 ? padicprec(x + 2, 2) %2 = +00 ? padicprec(2 + x + 0(x²), 2) %3 = +00 The function raises an exception if it encounters an object incompatible with *p*-adic computations:

```
? padicprec(0(3), 2)
 *** at top-level: padicprec(0(3),2)
 *** ^------
 *** padicprec: inconsistent moduli in padicprec: 3 != 2
? padicprec(1.0, 2)
 *** at top-level: padicprec(1.0,2)
 *** ^------
 *** padicprec: incorrect type in padicprec (t_REAL).
```

parapply(x)

Parallel evaluation of f on the elements of x. The function f must not access global variables or variables declared with local(), and must be free of side effects.

parapply(factor,[2^256 + 1, 2^193 - 1])

factors $2^{256} + 1$ and $2^{193} - 1$ in parallel.

```
{
  my(E = ellinit([1,3]), V = vector(12,i,randomprime(2^200)));
  parapply(p->ellcard(E,p), V)
}
```

computes the order of $E(\mathbb{F}_p)$ for 12 random primes of 200 bits.

pareval()

Parallel evaluation of the elements of x, where x is a vector of closures. The closures must be of arity 0, must not access global variables or variables declared with local and must be free of side effects.

Here is an artificial example explaining the MOV attack on the elliptic discrete log problem (by reducing it to a standard discrete log over a finite field):

```
{
    my(q = 2^30 + 3, m = 40 * q; p = 1 + m^2); \\ p, q are primes
    my(E = ellinit([0,0,0,1,0] * Mod(1,p)));
    my([P, Q] = ellgenerators(E));
    \\ E(F_p) ~ Z/m P + Z/m Q and the order of the
    \\ Weil pairing <P,Q> in (Z/p)^* is m
    my(F = [m,factor(m)], e = random(m), R, wR, wQ);
    R = ellpow(E, Q, e);
    wR = ellweilpairing(E,P,R,m);
    wQ = ellweilpairing(E,P,Q,m); \\ wR = wQ^e
    pareval([()->znlog(wR,wQ,F), ()->elllog(E,R,Q), ()->e])
}
```

Note the use of my to pass "arguments" to the functions we need to evaluate while satisfying the listed requirements: closures of arity 0 and no global variables (another possibility would be to use export). As a result, the final three statements satisfy all the listed requirements and are run in parallel. (Which is silly for this computation but illustrates the use of pareval.) The function parfor is more powerful but harder to use.

parselect(A, flag)

Selects elements of A according to the selection function f, done in parallel. If *flag* is 1, return the indices of those elements (indirect selection) The function f must not access global variables or variables declared with local(), and must be free of side effects.

permcycles()

Given a permutation x on n elements, return the orbits of 1, ..., n under the action of x as cycles.

```
? permcycles(Vecsmall([1,2,3]))
%1 = [Vecsmall([1]),Vecsmall([2]),Vecsmall([3])]
? permcycles(Vecsmall([2,3,1]))
%2 = [Vecsmall([1,2,3])]
? permcycles(Vecsmall([2,1,3]))
%3 = [Vecsmall([1,2]),Vecsmall([3])]
```

permorder()

Given a permutation x on n elements, return its order.

```
? p = Vecsmall([3,1,4,2,5]);
? p^2
%2 = Vecsmall([4,3,2,1,5])
? p^4
%3 = Vecsmall([1,2,3,4,5])
? permorder(p)
%4 = 4
```

permsign()

Given a permutation x on n elements, return its signature.

```
? p = Vecsmall([3,1,4,2,5]);
? permsign(p)
%2 = -1
? permsign(p^2)
%3 = 1
```

permtonum()

Given a permutation x on n elements, gives the number k such that x = numtoperm(n, k), i.e. inverse function of numtoperm. The numbering used is the standard lexicographic ordering, starting at 0.

plotdraw(flag)

Physically draw the rectwindow w. More generally, w can be of the form $[w_1, x_1, y_1, w_2, x_2, y_2, ...]$ (number of components must be divisible by 3; the windows w_1, w_2 , etc. are physically placed with their upper left corner at physical position $(x_1, y_1), (x_2, y_2), ...$ respectively, and are then drawn together. Overlapping regions will thus be drawn twice, and the windows are considered transparent. Then display the whole drawing in a window on your screen. If $flag! = 0, x_1, y_1$ etc. express fractions of the size of the current output device

plotexport(list, flag)

Draw list of rectwindows as in plotdraw(list,flag), returning the resulting picture as a character string which can then be written to a file. The format fmt is either "ps" (PostScript output) or "svg" (Scalable Vector Graphics).

```
? plotinit(0, 100, 100);
? plotbox(0, 50, 50);
? plotcolor(0, 2);
? plotbox(0, 30, 30);
? plotdraw(0); \\ watch result on screen
? s = plotexport("svg", 0);
? write("graph.svg", s); \\ dump result to file
```

plothraw(Y, flag)

Given X and Y two vectors of equal length, plots (in high precision) the points whose (x, y)-coordinates are given in X and Y. Automatic positioning and scaling is done, but with the same scaling factor on x and y. If flag is 1, join points, other nonzero flags toggle display options and should be combinations of bits 2^k , $k \ge 3$ as in ploth.

plothrawexport(X, Y, flag)

Given X and Y two vectors of equal length, plots (in high precision) the points whose (x, y)-coordinates are given in X and Y, returning the resulting picture as a character string which can then be written to a file. The format fmt is either "ps" (PostScript output) or "svg" (Scalable Vector Graphics).

Automatic positioning and scaling is done, but with the same scaling factor on x and y. If flag is 1, join points, other nonzero flags toggle display options and should be combinations of bits 2^k , $k \ge 3$ as in ploth.

polclass(inv, x)

Return a polynomial in $\mathbb{Z}[x]$ generating the Hilbert class field for the imaginary quadratic discriminant D. If *inv* is 0 (the default), use the modular *j*-function and return the classical Hilbert polynomial, otherwise use a class invariant. The following invariants correspond to the different values of *inv*, where *f* denotes Weber's function weber, and $w_{p,q}$ the double eta quotient given by $w_{p,q} = (\eta(x/p)\eta(x/q))/(\eta(x)\eta(x/pq))$

The invariants $w_{p,q}$ are not allowed unless they satisfy the following technical conditions ensuring they do generate the Hilbert class field and not a strict subfield:

- if p! = q, we need them both noninert, prime to the conductor of $\mathbb{Z}[\sqrt{D}]$. Let P, Q be prime ideals above p and q; if both are unramified, we further require that P^1Q^1 be all distinct in the class group of $\mathbb{Z}[\sqrt{D}]$; if both are ramified, we require that PQ! = 1 in the class group.
- if p = q, we want it split and prime to the conductor and the prime ideal above it must have order ! = 1, 2, 4 in the class group.

Invariants are allowed under the additional conditions on D listed below.

- 0: j
- $1: f, D = 1 \mod 8$ and $D = 1, 2 \mod 3;$
- $2: f^2, D = 1 \mod 8$ and $D = 1, 2 \mod 3;$
- $3: f^3, D = 1 \mod 8;$
- $4: f^4, D = 1 \mod 8$ and $D = 1, 2 \mod 3;$
- $5: \gamma_2 = j^{1/3}, D = 1, 2mod3;$
- $6: w_{2,3}, D = 1 \mod 8$ and $D = 1, 2 \mod 3$;
- 8: f^8 , D = 1mod8 and D = 1, 2mod3;
- 9: $w_{3,3}$, $D = 1 \mod 2$ and $D = 1, 2 \mod 3$;
- 10: $w_{2,5}$, $D! = 60 \mod 80$ and $D = 1, 2 \mod 3$;
- 14: $w_{2,7}, D = 1 \mod 8;$
- 15: $w_{3,5}$, D = 1, 2mod3;
- 21: $w_{3,7}$, $D = 1 \mod 2$ and 21 does not divide D
- 23: $w_{2,3}^2$, D = 1, 2mod3;
- 24: $w_{2.5}^2$, D = 1, 2mod3;
- 26: $w_{2,13}, D! = 156 mod 208;$
- 27: $w_{2,7}^2$, D! = 28mod112;

- 28: $w_{3,3}^2$, D = 1, 2mod3;
- 35: $w_{5,7}$, D = 1, 2mod3;
- 39: $w_{3,13}$, $D = 1 \mod 2$ and $D = 1, 2 \mod 3$;

The algorithm for computing the polynomial does not use the floating point approach, which would evaluate a precise modular function in a precise complex argument. Instead, it relies on a faster Chinese remainder based approach modulo small primes, in which the class invariant is only defined algebraically by the modular polynomial relating the modular function to j. So in fact, any of the several roots of the modular polynomial may actually be the class invariant, and more precise assertions cannot be made.

For instance, while polclass(D) returns the minimal polynomial of $j(\tau)$ with τ (any) quadratic integer for the discriminant D, the polynomial returned by polclass(D, 5) can be the minimal polynomial of any of $\gamma_2(\tau)$, $\zeta_3\gamma_2(\tau)$ or $\zeta_3^2\gamma_2(\tau)$, the three roots of the modular polynomial $j = \gamma_2^3$, in which j has been specialised to $j(\tau)$.

The modular polynomial is given by $j = ((f^{24} - 16)^3)/(f^{24})$ for Weber's function f.

For the double eta quotients of level N = pq, all functions are covered such that the modular curve $X_0^+(N)$, the function field of which is generated by the functions invariant under $\Gamma^0(N)$ and the Fricke-Atkin-Lehner involution, is of genus 0 with function field generated by (a power of) the double eta quotient w. This ensures that the full Hilbert class field (and not a proper subfield) is generated by class invariants from these double eta quotients. Then the modular polynomial is of degree 2 in j, and of degree $\psi(N) = (p+1)(q+1)$ in w.

```
? polclass(-163)
%1 = x + 262537412640768000
? polclass(-51, , 'z)
%2 = z^2 + 5541101568*z + 6262062317568
? polclass(-151,1)
x^7 - x^6 + x^5 + 3*x^3 - x^2 + 3*x + 1
```

polcoef(*n*, *v*)

Coefficient of degree n of the polynomial x, with respect to the main variable if v is omitted, with respect to v otherwise. If n is greater than the degree, the result is zero.

Naturally applies to scalars (polynomial of degree 0), as well as to rational functions whose denominator is a monomial. It also applies to power series: if n is less than the valuation, the result is zero. If it is greater than the largest significant degree, then an error message is issued.

polcoeff(n, v)

Deprecated alias for polcoef.

polcompositum(Q, flag)

P and *Q* being squarefree polynomials in $\mathbb{Z}[X]$ in the same variable, outputs the simple factors of the étale \mathbb{Q} -algebra $A = \mathbb{Q}(X, Y)/(P(X), Q(Y))$. The factors are given by a list of polynomials *R* in $\mathbb{Z}[X]$, attached to the number field $\mathbb{Q}(X)/(R)$, and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. In this case, there will be a single factor R if and only if the number fields defined by P and Q are linearly disjoint (their intersection is \mathbb{Q}).

Assuming P is irreducible (of smaller degree than Q for efficiency), it is in general much faster to proceed as follows

```
nf = nfinit(P); L = nffactor(nf, Q)[,1];
vector(#L, i, rnfequation(nf, L[i]))
```

to obtain the same result. If you are only interested in the degrees of the simple factors, the rnfequation instruction can be replaced by a trivial poldegree(P) * poldegree(L[i]).

The binary digits of flag mean

1: outputs a vector of 4-component vectors [R, a, b, k], where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of $\mathbb{Q}(X)/(R)$. Finally, k is a small integer such that b + ka = X modulo R.

2: assume that P and Q define number fields which are linearly disjoint: both polynomials are irreducible and the corresponding number fields have no common subfield besides \mathbb{Q} . This allows to save a costly factorization over \mathbb{Q} . In this case return the single simple factor instead of a vector with one element.

A compositum is often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is an example involving the field $\mathbb{Q}(\zeta_5, 5^{1/5})$:

```
? L = polcompositum(x^5 - 5, polcyclo(5), 1); \\ list of [R,a,b,k]
? [R, a] = L[1]; \\ pick the single factor, extract R, a (ignore b,k)
? R \ defines the compositum
3 = x^{20} + 5x^{19} + 15x^{18} + 35x^{17} + 70x^{16} + 141x^{15} + 260x^{14}
+ 355*x^13 + 95*x^12 - 1460*x^11 - 3279*x^10 - 3660*x^9 - 2005*x^8 \
+ 705*x^7 + 9210*x^6 + 13506*x^5 + 7145*x^4 - 2740*x^3 + 1040*x^2
- 320*x + 256
? a^5 - 5 \setminus a fifth root of 5
\%4 = 0
? [T, X] = polredbest(R, 1);
? T \\ simpler defining polynomial for Q[x]/(R)
\%6 = x^20 + 25 x^10 + 5
? X \\ root of R in Q[y]/(T(y))
\%7 = Mod(-1/11*x^{15} - 1/11*x^{14} + 1/22*x^{10} - 47/22*x^{5} - 29/11*x^{4} + 7/22,)
x^{20} + 25 * x^{10} + 5
? a = subst(a.pol, 'x, X) \setminus a in the new coordinates
\%8 = Mod(1/11*x^{14} + 29/11*x^{4}, x^{20} + 25*x^{10} + 5)
? a^5 - 5
%9 = 0
```

In the above example, $x^5 - 5$ and the 5-th cyclotomic polynomial are irreducible over \mathbb{Q} ; they have coprime degrees so define linearly disjoint extensions and we could have started by

? [R,a] = polcompositum(x⁵ - 5, polcyclo(5), 3); \\ [R,a,b,k]

polcyclofactors()

Returns a vector of polynomials, whose product is the product of distinct cyclotomic polynomials dividing f.

```
? f = x^10+5*x^8-x^7+8*x^6-4*x^5+8*x^4-3*x^3+7*x^2+3;
? v = polcyclofactors(f)
%2 = [x^2 + 1, x^2 + x + 1, x^4 - x^3 + x^2 - x + 1]
? apply(poliscycloprod, v)
%3 = [1, 1, 1]
? apply(poliscyclo, v)
%4 = [4, 3, 10]
```

In general, the polynomials are products of cyclotomic polynomials and not themselves irreducible:

```
? g = x^8+2*x^7+6*x^6+9*x^5+12*x^4+11*x^3+10*x^2+6*x+3;
? polcyclofactors(g)
```

```
\%2 = [x^{6} + 2*x^{5} + 3*x^{4} + 3*x^{3} + 3*x^{2} + 2*x + 1]
? factor(%[1])
%3 =
[ x^{2} + x + 1 1]
[x^{4} + x^{3} + x^{2} + x + 1 1]
```

poldegree(v)

Degree of the polynomial x in the main variable if v is omitted, in the variable v otherwise.

The degree of 0 is -oo. The degree of a nonzero scalar is 0. Finally, when x is a nonzero polynomial or rational function, returns the ordinary degree of x. Raise an error otherwise.

poldisc(v)

Discriminant of the polynomial *pol* in the main variable if v is omitted, in v otherwise. Uses a modular algorithm over \mathbb{Z} or \mathbb{Q} , and the subresultant algorithm otherwise.

```
? T = x^4 + 2*x+1;
? poldisc(T)
%2 = -176
? poldisc(T^2)
%3 = 0
```

For convenience, the function also applies to types t_QUAD and t_QFI/t_QFR:

```
? z = 3*quadgen(8) + 4;
? poldisc(z)
%2 = 8
? q = Qfb(1,2,3);
? poldisc(q)
%4 = -8
```

poldiscfactors(flag)

Given a polynomial T with integer coefficients, return [D, faD] where D is the discriminant of T and faD is a cheap partial factorization of ||D||: entries in its first column are coprime and not perfect powers but need not be primes. The factors are obtained by a combination of trial division, testing for perfect powers, factorizations in coprimes, and computing Euclidean remainder sequences for (T, T') modulo composite factors d of D (which is likely to produce 0-divisors in $\mathbb{Z}/d\mathbb{Z}$). If *flag* is 1, finish the factorization using factorint.

```
? T = x^3 - 6021021*x^2 + 12072210077769*x - 8092423140177664432;
? [D,faD] = poldiscfactors(T); print(faD); D
[3, 3; 7, 2; 373, 2; 500009, 2; 24639061, 2]
%2 = -27937108625866859018515540967767467
? T = x^3 + 9*x^2 + 27*x - 125014250689643346789780229390526092263790263725;
? [D,faD] = poldiscfactors(T); print(faD)
[2, 6; 3, 3; 125007125141751093502187, 4]
? [D,faD] = poldiscfactors(T, 1); print(faD)
[2, 6; 3, 3; 500009, 12; 1000003, 4]
```

poldiscreduced()

Reduced discriminant vector of the (integral, monic) polynomial f. This is the vector of elementary divisors of $\mathbb{Z}[\alpha]/f'(\alpha)\mathbb{Z}[\alpha]$, where α is a root of the polynomial f. The components of the result are all positive, and their product is equal to the absolute value of the discriminant of f.

polgalois(precision)

Galois group of the nonconstant polynomial $T \in \mathbb{Q}[X]$. In the present version **2.13.3**, T must be irreducible and the degree d of T must be less than or equal to 7. If the galdata package has been installed, degrees 8, 9, 10 and 11 are also implemented. By definition, if $K = \mathbb{Q}[x]/(T)$, this computes the action of the Galois group of the Galois closure of K on the d distinct roots of T, up to conjugacy (corresponding to different root orderings).

The output is a 4-component vector [n, s, k, name] with the following meaning: n is the cardinality of the group, s is its signature (s = 1 if the group is a subgroup of the alternating group A_d , s = -1 otherwise) and name is a character string containing name of the transitive group according to the GAP 4 transitive groups library by Alexander Hulpke.

k is more arbitrary and the choice made up to version 2.2.3 of PARI is rather unfortunate: for d > 7, k is the numbering of the group among all transitive subgroups of S_d , as given in "The transitive groups of degree up to eleven", G. Butler and J. McKay, *Communications in Algebra*, vol. 11, 1983, pp. 863–911 (group k is denoted T_k there). And for d <= 7, it was ad hoc, so as to ensure that a given triple would denote a unique group. Specifically, for polynomials of degree d <= 7, the groups are coded as follows, using standard notations

- In degree 1: $S_1 = [1, 1, 1]$.
- In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1], S_3 = [6, -1, 1].$

In degree 4: $C_4 = [4, -1, 1], V_4 = [4, 1, 1], D_4 = [8, -1, 1], A_4 = [12, 1, 1], S_4 = [24, -1, 1].$

In degree 5: $C_5 = [5, 1, 1], D_5 = [10, 1, 1], M_{20} = [20, -1, 1], A_5 = [60, 1, 1], S_5 = [120, -1, 1].$

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 1]$, $A_4 = [12, 1, 1]$, $G_{18} = [18, -1, 1]$, $S_4^- = [24, -1, 1]$, $A_4xC_2 = [24, -1, 2]$, $S_4^+ = [24, 1, 1]$, $G_{36}^- = [36, -1, 1]$, $G_{36}^+ = [36, 1, 1]$, $S_4xC_2 = [48, -1, 1]$, $A_5 = PSL_2(5) = [60, 1, 1]$, $G_{72} = [72, -1, 1]$, $S_5 = PGL_2(5) = [120, -1, 1]$, $A_6 = [360, 1, 1]$, $S_6 = [720, -1, 1]$.

In degree 7: $C_7 = [7, 1, 1], D_7 = [14, -1, 1], M_{21} = [21, 1, 1], M_{42} = [42, -1, 1], PSL_2(7) = PSL_3(2) = [168, 1, 1], A_7 = [2520, 1, 1], S_7 = [5040, -1, 1].$

This is deprecated and obsolete, but for reasons of backward compatibility, we cannot change this behavior yet. So you can use the default new_galois_format to switch to a consistent naming scheme, namely k is always the standard numbering of the group among all transitive subgroups of S_n . If this default is in effect, the above groups will be coded as:

In degree 1:
$$S_1 = [1, 1, 1]$$
.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1], S_3 = [6, -1, 2].$

In degree 4: $C_4 = [4, -1, 1], V_4 = [4, 1, 2], D_4 = [8, -1, 3], A_4 = [12, 1, 4], S_4 = [24, -1, 5].$

In degree 5: $C_5 = [5, 1, 1], D_5 = [10, 1, 2], M_{20} = [20, -1, 3], A_5 = [60, 1, 4], S_5 = [120, -1, 5].$

In degree 6: $C_6 = [6, -1, 1], S_3 = [6, -1, 2], D_6 = [12, -1, 3], A_4 = [12, 1, 4], G_{18} = [18, -1, 5], A_4xC_2 = [24, -1, 6], S_4^+ = [24, 1, 7], S_4^- = [24, -1, 8], G_{36}^- = [36, -1, 9], G_{36}^+ = [36, 1, 10], S_4xC_2 = [48, -1, 11], A_5 = PSL_2(5) = [60, 1, 12], G_{72} = [72, -1, 13], S_5 = PGL_2(5) = [120, -1, 14], A_6 = [360, 1, 15], S_6 = [720, -1, 16].$

In degree 7: $C_7 = [7, 1, 1]$, $D_7 = [14, -1, 2]$, $M_{21} = [21, 1, 3]$, $M_{42} = [42, -1, 4]$, $PSL_2(7) = PSL_3(2) = [168, 1, 5]$, $A_7 = [2520, 1, 6]$, $S_7 = [5040, -1, 7]$.

Warning. The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

polgraeffe()

Returns the Graeffe transform g of f, such that $g(x^2) = f(x)f(-x)$.

polhensellift(B, p, e)

Given a prime p, an integral polynomial A whose leading coefficient is a p-unit, a vector B of integral polynomials that are monic and pairwise relatively prime modulo p, and whose product is congruent to A/lc(A) modulo p, lift the elements of B to polynomials whose product is congruent to A modulo p^e .

More generally, if T is an integral polynomial irreducible mod p, and B is a factorization of A over the finite field $\mathbb{F}_p[t]/(T)$, you can lift it to $\mathbb{Z}_p[t]/(T, p^e)$ by replacing the p argument with [p, T]:

? { T = t³ - 2; p = 7; A = x² + t + 1; B = [x + (3*t² + t + 1), x + (4*t² + 6*t + 6)]; r = polhensellift(A, B, [p, T], 6) } %1 = [x + (20191*t² + 50604*t + 75783), x + (97458*t² + 67045*t + 41866)] ? liftall(r[1] * r[2] * Mod(Mod(1,p⁶),T)) %2 = x² + (t + 1)

polinterpolate(Y, t, e)

Given the data vectors X and Y of the same length n (X containing the x-coordinates, and Y the corresponding y-coordinates), this function finds the interpolating polynomial P of minimal degree passing through these points and evaluates it at t. If Y is omitted, the polynomial P interpolates the (i, X[i]).

```
? v = [1, 2, 4, 8, 11, 13];
? P = polinterpolate(v) \\ formal interpolation
%1 = 7/120*x^5 - 25/24*x^4 + 163/24*x^3 - 467/24*x^2 + 513/20*x - 11
? [ subst(P, 'x,a) | a <- [1..6] ]
%2 = [1, 2, 4, 8, 11, 13]
? polinterpolate(v, 10) \\ evaluate at 10
%3 = 508
? subst(P, x, 10)
%4 = 508
? P = polinterpolate([1,2,4], [9,8,7])
%5 = 1/6*x^2 - 3/2*x + 31/3
? [subst(P, 'x, a) | a <- [1,2,4]]
%6 = [9, 8, 7]
? P = polinterpolate([1,2,4], [9,8,7], 0)
%7 = 31/3
```

If the goal is to extrapolate a function at a unique point, it is more efficient to use the t argument rather than interpolate formally then evaluate:

```
? x0 = 1.5;
? v = vector(20, i,random([-10,10]));
? for(i=1,10^3, subst(polinterpolate(v),'x, x0))
time = 352 ms.
? for(i=1,10^3, polinterpolate(v,,x0))
time = 111 ms.
? v = vector(40, i,random([-10,10]));
? for(i=1,10^3, subst(polinterpolate(v), 'x, x0))
time = 3,035 ms.
? for(i=1,10^3, polinterpolate(v,, x0))
time = 436 ms.
```

The threshold depends on the base field. Over small prime finite fields, interpolating formally first is more efficient

```
? bench(p, N, T = 10^3) =
{ my (v = vector(N, i, random(Mod(0,p))));
 my (x0 = Mod(3, p), t1, t2);
gettime();
for(i=1, T, subst(polinterpolate(v), 'x, x0));
t1 = gettime();
for(i=1, T, polinterpolate(v,, x0));
t2 = gettime(); [t1, t2];
}
? p = 101;
? bench(p, 4, 10^4) \\ both methods are equivalent
%3 = [39, 40]
? bench(p, 40) \\ with 40 points formal is much faster
%4 = [45, 355]
```

As the cardinality increases, formal interpolation requires more points to become interesting:

```
? p = nextprime(2^{128});
? bench(p, 4) \setminus formal is slower
\%3 = [16, 9]
? bench(p, 10) \setminus formal has become faster
%4 = [61, 70]
? bench(p, 100) \setminus formal is much faster
\%5 = [1682, 9081]
? p = nextprime(10^{500});
? bench(p, 4) \setminus formal is slower
\%7 = [72, 354]
? bench(p, 20) \\ formal is still slower
\%8 = [1287, 962]
? bench(p, 40) \setminus formal has become faster
\%9 = [3717, 4227]
? bench(p, 100) \\ faster but relatively less impressive
\%10 = [16237, 32335]
```

If t is a complex numeric value and e is present, e will contain an error estimate on the returned value. More precisely, let P be the interpolation polynomial on the given n points; there exist a subset of n - 1 points and Q the attached interpolation interpolation polynomial such that e = exponent(P(t) - Q(t)) (Neville's algorithm).

```
? f(x) = 1 / (1 + 25*x^2);
? x0 = 975/1000;
? test(X) =
  { my (P, e);
P = polinterpolate(X, [f(x) | x <- X], x0, &e);
  [ exponent(P - f(x0)), e ];
  }
  \\ equidistant nodes vs. Chebyshev nodes
? test( [-10..10] / 10 )
%4 = [6, 5]
? test( polrootsreal(polchebyshev(21)) )
%5 = [-15, -10]
```

```
? test( [-100..100] / 100 )
%7 = [93, 97] \\ P(x0) is way different from f(x0)
? test( polrootsreal(polchebyshev(201)) )
%8 = [-60, -55]
```

This is an example of Runge's phenomenon: increasing the number of equidistant nodes makes extrapolation much worse. Note that the error estimate is not a guaranteed upper bound (cf %4), but is reasonably tight in practice.

poliscyclo()

Returns 0 if f is not a cyclotomic polynomial, and n > 0 if $f = \Phi_n$, the n-th cyclotomic polynomial.

```
? poliscyclo(x^4-x^2+1)
%1 = 12
? polcyclo(12)
%2 = x^4 - x^2 + 1
? poliscyclo(x^4-x^2-1)
%3 = 0
```

poliscycloprod()

Returns 1 if f is a product of cyclotomic polynomial, and 0 otherwise.

```
? f = x^6+x^5-x^3+x+1;
? poliscycloprod(f)
%2 = 1
? factor(f)
%3 =
[ x^2 + x + 1 1]
[x^4 - x^2 + 1 1]
? [ poliscyclo(T) | T <- %[,1] ]
%4 = [3, 12]
? polcyclo(3) * polcyclo(12)
%5 = x^6 + x^5 - x^3 + x + 1
```

polisirreducible()

pol being a polynomial (univariate in the present version **2.13.3**), returns 1 if *pol* is nonconstant and irreducible, 0 otherwise. Irreducibility is checked over the smallest base field over which *pol* seems to be defined.

pollead(v)

Leading coefficient of the polynomial or power series x. This is computed with respect to the main variable of x if v is omitted, with respect to the variable v otherwise.

polrecip()

Reciprocal polynomial of *pol* with respect to its main variable, i.e. the coefficients of the result are in reverse order; *pol* must be a polynomial.

? polrecip(x² + 2*x + 3) %1 = 3*x² + 2*x + 1 ? polrecip(2*x + y) %2 = y*x + 2

polred(flag, _arg2)

This function is deprecated, use polredbest instead. Finds polynomials with reasonably small coefficients

defining subfields of the number field defined by T. One of the polynomials always defines \mathbb{Q} (hence has degree 1), and another always defines the same number field as T if T is irreducible.

All T accepted by nfinit are also allowed here; in particular, the format [T, listP] is recommended, e.g. with $listP = 10^5$ or a vector containing all ramified primes. Otherwise, the maximal order of $\mathbb{Q}[x]/(T)$ must be computed.

The following binary digits of flag are significant:

1: Possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than primelimit or divide integers stored in the addprimes table. This flag is *deprecated*, the [T, listP] format is more flexible.

2: gives also elements. The result is a two-column matrix, the first column giving primitive elements defining these subfields, the second giving the corresponding minimal polynomials.

```
? M = polred(x^4 + 8, 2)
%1 =
[ 1 x - 1]
[ 1/2*x^2 + 1 x^2 - 2*x + 3]
[-1/2*x^2 + 1 x^2 - 2*x + 3]
[ 1/2*x^2 x^2 + 2]
[ 1/4*x^3 x^4 + 2]
? minpoly(Mod(M[2,1], x^4+8))
%2 = x^2 + 2
```

polredabs(flag)

Returns a canonical defining polynomial P for the number field $\mathbb{Q}[X]/(T)$ defined by T, such that the sum of the squares of the modulus of the roots (i.e. the T_2 -norm) is minimal. Different T defining isomorphic number fields will yield the same P. All T accepted by nfinit are also allowed here, e.g. nonmonic polynomials, or pairs [T, listP] specifying that a nonmaximal order may be used. For convenience, any number field structure (nf, bnf, ...) can also be used instead of T.

? polredabs(x² + 16) %1 = x² + 1 ? K = bnfinit(x² + 16); polredabs(K) %2 = x² + 1

Warning 1. Using a t_POL T requires computing and fully factoring the discriminant d_K of the maximal order which may be very hard. You can use the format [T, listP], where listP encodes a list of known coprime divisors of disc(T) (see ??nfbasis), to help the routine, thereby replacing this part of the algorithm by a polynomial time computation But this may only compute a suborder of the maximal order, when the divisors are not squarefree or do not include all primes dividing d_K . The routine attempts to certify the result independently of this order computation as per nfcertify: we try to prove that the computed order is maximal. If the certification fails, the routine then fully factors the integers returned by nfcertify. You can also use polredbest to avoid this factorization step; in this case, the result is small but no longer canonical.

Warning 2. Apart from the factorization of the discriminant of T, this routine runs in polynomial time for a *fixed* degree. But the complexity is exponential in the degree: this routine may be exceedingly slow when the number field has many subfields, hence a lot of elements of small T_2 -norm. If you do not need a canonical polynomial, the function **polredbest** is in general much faster (it runs in polynomial time), and tends to return polynomials with smaller discriminants.

The binary digits of flag mean

1: outputs a two-component row vector [P, a], where P is the default output and Mod(a, P) is a root of the original T.

4: gives all polynomials of minimal T_2 norm; of the two polynomials P(x) and P(-x), only one is given.

16: (OBSOLETE) Possibly use a suborder of the maximal order, *without* attempting to certify the result as in Warning 1. This makes polredbas behave like polredbest. Just use the latter.

```
? T = x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 \
        - 7453176*x^6 + 13950764*x^4 - 5596840*x^2 + 46225
? T1 = polredbs(T); T2 = polredbest(T);
? [ norml2(polroots(T1)), norml2(polroots(T2)) ]
%3 = [88.0000000, 120.000000]
? [ sizedigit(poldisc(T1)), sizedigit(poldisc(T2)) ]
%4 = [75, 67]
```

The precise definition of the output of polredabs is as follows.

- Consider the finite list of characteristic polynomials of primitive elements of K that are in \mathbb{Z}_K and minimal for the T_2 norm; now remove from the list the polynomials whose discriminant do not have minimal absolute value. Note that this condition is restricted to the original list of polynomials with minimal T_2 norm and does not imply that the defining polynomial for the field with smallest discriminant belongs to the list !
- To a polynomial $P(x) = x^n + ... + a_n \in \mathbb{R}[x]$ we attach the sequence S(P) given by $||a_1||, a_1, ..., ||a_n||, a_n$. Order the polynomials P by the lexicographic order on the coefficient vectors S(P). Then the output of polredabs is the smallest polynomial in the above list for that order. In other words, the monic polynomial which is lexicographically smallest with respect to the absolute values of coefficients, favouring negative coefficients to break ties, i.e. choosing $x^3 - 2$ rather than $x^3 + 2$.

polredbest(flag)

Finds a polynomial with reasonably small coefficients defining the same number field as *T*. All *T* accepted by nfinit are also allowed here (e.g. nonmonic polynomials, nf, bnf, [T,Z_K_basis]). Contrary to polredabs, this routine runs in polynomial time, but it offers no guarantee as to the minimality of its result.

This routine computes an LLL-reduced basis for an order in $\mathbb{Q}[X]/(T)$, then examines small linear combinations of the basis vectors, computing their characteristic polynomials. It returns the *separable* polynomial P of smallest discriminant, the one with lexicographically smallest abs(Vec(P)) in case of ties. This is a good candidate for subsequent number field computations since it guarantees that the denominators of algebraic integers, when expressed in the power basis, are reasonably small. With no claim of minimality, though.

It can happen that iterating this functions yields better and better polynomials, until it stabilizes:

```
? \p5
? P = X^12+8*X^8-50*X^6+16*X^4-3069*X^2+625;
? poldisc(P)*1.
%2 = 1.2622 E55
? P = polredbest(P);
? poldisc(P)*1.
%4 = 2.9012 E51
? P = polredbest(P);
? poldisc(P)*1.
%6 = 8.8704 E44
```

In this example, the initial polynomial P is the one returned by polredabs, and the last one is stable.

If flag = 1: outputs a two-component row vector [P, a], where P is the default output and Mod(a, P) is a root of the original T.

? [P,a] = polredbest(x⁴ + 8, 1) %1 = [x⁴ + 2, Mod(x³, x⁴ + 2)] ? charpoly(a) %2 = x⁴ + 8

In particular, the map $\mathbb{Q}[x]/(T) \to \mathbb{Q}[x]/(P)$, x : -- > Mod(a, P) defines an isomorphism of number fields, which can be computed as

subst(lift(Q), 'x, a)

if Q is a t_POLMOD modulo T; b = modreverse(a) returns a t_POLMOD giving the inverse of the above map (which should be useless since $\mathbb{Q}[x]/(P)$ is a priori a better representation for the number field and its elements).

polredord()

This function is obsolete, use polredbest.

polresultant(y, v, flag)

Resultant of the two polynomials x and y with exact entries, with respect to the main variables of x and y if v is omitted, with respect to the variable v otherwise. The algorithm assumes the base ring is a domain. If you also need the u and v such that x * u + y * v = Res(x, y), use the polresultantext function.

If flag = 0 (default), uses the algorithm best suited to the inputs, either the subresultant algorithm (Lazard/Ducos variant, generic case), a modular algorithm (inputs in $\mathbb{Q}[X]$) or Sylvester's matrix (inexact inputs).

If flag = 1, uses the determinant of Sylvester's matrix instead; this should always be slower than the default.

If x or y are multivariate with a huge *polynomial* content, it is advisable to remove it before calling this function. Compare:

? a = polcyclo(7) * ((t+1)/(t+2))^100; ? b = polcyclo(11)* ((t+2)/(t+3))^100); ? polresultant(a,b); time = 3,833 ms. ? ca = content(a); cb = content(b); \ polresultant(a/ca,b/cb)*ca^poldegree(b)*cb*poldegree(a); \\ instantaneous

The function only removes rational denominators and does not compute automatically the content because it is generically small and potentially *very* expensive (e.g. in multivariate contexts). The choice is yours, depending on your application.

polresultantext(B, v)

Finds polynomials U and V such that A * U + B * V = R, where R is the resultant of U and V with respect to the main variables of A and B if v is omitted, and with respect to v otherwise. Returns the row vector [U, V, R]. The algorithm used (subresultant) assumes that the base ring is a domain.

? A = x*y; B = (x+y)^2; ? [U,V,R] = polresultantext(A, B) %2 = [-y*x - 2*y^2, y^2, y^4] ? A*U + B*V %3 = y^4 ? [U,V,R] = polresultantext(A, B, y) %4 = [-2*x^2 - y*x, x^2, x^4] ? A*U+B*V %5 = x^4

polroots(precision)

Complex roots of the polynomial T, given as a column vector where each root is repeated according to its multiplicity and given as floating point complex numbers at the current realprecision:

? polroots(x^2)
%1 = [0.E-38 + 0.E-38*I, 0.E-38 + 0.E-38*I]~
? polroots(x^3+1)
%2 = [-1.00... + 0.E-38*I, 0.50... - 0.866...*I, 0.50... + 0.866...*I]~

The algorithm used is a modification of Schönhage's root-finding algorithm, due to and originally implemented by Gourdon. It runs in polynomial time in deg(T) and the precision. If furthermore T has rational coefficients, roots are guaranteed to the required relative accuracy. If the input polynomial T is exact, then the ordering of the roots does not depend on the precision: they are ordered by increasing $||\Im z||$, then by increasing $\Re z$; in case of tie (conjugates), the root with negative imaginary part comes first.

polrootsbound(tau)

Return a sharp upper bound B for the modulus of the largest complex root of the polynomial T with complex coefficients with relative error τ . More precisely, we have $||z|| \le B$ for all roots and there exist one root such that $||z_0|| \ge B \exp(-2\tau)$. Much faster than either polynomials or polynotic polynomial.

```
? T=poltchebi(500);
? vecmax(abs(polroots(T)))
time = 5.706 ms.
%2 = 0.99999506520185816611184481744870013191
? vecmax(abs(polrootsreal(T)))
time = 1,972 ms.
%3 = 0.99999506520185816611184481744870013191
? polrootsbound(T)
time = 217 \text{ ms}.
\%4 = 1.0098792554165905155
? polrootsbound(T, log(2)/2) \\ allow a factor 2, much faster
time = 51 \text{ ms}.
\%5 = 1.4065759938190154354
? polrootsbound(T, 1e-4)
time = 504 \text{ ms}.
\%6 = 1.0000920717983847741
? polrootsbound(T, 1e-6)
time = 810 \text{ ms}.
%7 = 0.9999960628901692905
? polrootsbound(T, 1e-10)
time = 1,351 ms.
\%8 = 0.9999950652993869760
```

polrootsff(p, a)

Obsolete, kept for backward compatibility: use factormod.

polrootsmod(D)

Vector of roots of the polynomial f over the finite field defined by the domain D as follows:

- D = p a prime: factor over \mathbb{F}_p ;
- D = [T, p] for a prime p and T(y) an irreducible polynomial over \mathbb{F}_p : factor over $\mathbb{F}_p[y]/(T)$ (as usual the main variable of T must have lower priority than the main variable of f);
- *D* a t_FFELT: factor over the attached field;

• D omitted: factor over the field of definition of f, which must be a finite field.

Multiple roots are not repeated.

```
? polrootsmod(x^2-1,2)
%1 = [Mod(1, 2)]~
? polrootsmod(x^2+1,3)
%2 = []~
? polrootsmod(x^2+1, [y^2+1,3])
%3 = [Mod(Mod(1, 3)*y, Mod(1, 3)*y^2 + Mod(1, 3)),
Mod(Mod(2, 3)*y, Mod(1, 3)*y^2 + Mod(1, 3))]~
? polrootsmod(x^2 + Mod(1,3))
%4 = []~
? liftall( polrootsmod(x^2 + Mod(Mod(1,3),y^2+1)) )
%5 = [y, 2*y]~
? t = ffgen(y^2+Mod(1,3)); polrootsmod(x^2 + t^0)
%6 = [y, 2*y]~
```

polrootspadic(p, r)

Vector of *p*-adic roots of the polynomial *pol*, given to *p*-adic precision *r*; the integer *p* is assumed to be a prime. Multiple roots are *not* repeated. Note that this is not the same as the roots in $\mathbb{Z}/p^r\mathbb{Z}$, rather it gives approximations in $\mathbb{Z}/p^r\mathbb{Z}$ of the true roots living in \mathbb{Q}_p :

```
? polrootspadic(x^3 - x^2 + 64, 2, 4)
%1 = [2^3 + 0(2^4), 2^3 + 0(2^4), 1 + 0(2^4)]~
? polrootspadic(x^3 - x^2 + 64, 2, 5)
%2 = [2^3 + 0(2^5), 2^3 + 2^4 + 0(2^5), 1 + 0(2^5)]~
```

As the second commands show, the first two roots *are* distinct in \mathbb{Q}_p , even though they are equal modulo 2^4 .

More generally, if T is an integral polynomial irreducible mod p and f has coefficients in $\mathbb{Q}[t]/(T)$, the argument p may be replaced by the vector [T, p]; we then return the roots of f in the unramified extension $\mathbb{Q}_p[t]/(T)$.

```
? polrootspadic(x<sup>3</sup> - x<sup>2</sup> + 64*y, [y<sup>2</sup>+y+1,2], 5)
%3 = [Mod((2<sup>3</sup> + 0(2<sup>5</sup>))*y + (2<sup>3</sup> + 0(2<sup>5</sup>)), y<sup>2</sup> + y + 1),
Mod((2<sup>3</sup> + 2<sup>4</sup> + 0(2<sup>5</sup>))*y + (2<sup>3</sup> + 2<sup>4</sup> + 0(2<sup>5</sup>)), y<sup>2</sup> + y + 1),
Mod(1 + 0(2<sup>5</sup>), y<sup>2</sup> + y + 1)]~
```

If *pol* has inexact t_PADIC coefficients, this need not well-defined; in this case, the polynomial is first made integral by dividing out the *p*-adic content, then lifted to \mathbb{Z} using truncate coefficientwise. Hence the roots given are approximations of the roots of an exact polynomial which is *p*-adically close to the input. To avoid pitfalls, we advise to only factor polynomials with exact rational coefficients.

polrootsreal(ab, precision)

Real roots of the polynomial T with real coefficients, multiple roots being included according to their multiplicity. If the polynomial does not have rational coefficients, it is first rescaled and rounded. The roots are given to a relative accuracy of realprecision. If argument ab is present, it must be a vector [a, b] with two components (of type t_INT, t_FRAC or t_INFINITY) and we restrict to roots belonging to that closed interval.

? \p9
? polrootsreal(x^2-2)
%1 = [-1.41421356, 1.41421356]~
? polrootsreal(x^2-2, [1,+00])
%2 = [1.41421356]~
? polrootsreal(x^2-2, [2,3])

```
%3 = []~
? polrootsreal((x-1)*(x-2), [2,3])
%4 = [2.00000000]~
```

The algorithm used is a modification of Uspensky's method (relying on Descartes's rule of sign), following Rouillier and Zimmerman's article "Efficient isolation of a polynomial real roots" (http://hal.inria.fr/ inria-00072518/). Barring bugs, it is guaranteed to converge and to give the roots to the required accuracy.

Remark. If the polynomial T is of the form $Q(x^h)$ for some $h \ge 2$ and ab is omitted, the routine will apply the algorithm to Q (restricting to nonnegative roots when h is even), then take h-th roots. On the other hand, if you want to specify ab, you should apply the routine to Q yourself and a suitable interval [a', b'] using approximate h-th roots adapted to your problem: the function will not perform this change of variables if ab is present.

polsturm(ab, _arg2)

Number of distinct real roots of the real polynomial T. If the argument ab is present, it must be a vector [a, b] with two real components (of type t_INT, t_REAL, t_FRAC or t_INFINITY) and we count roots belonging to that closed interval.

If possible, you should stick to exact inputs, that is avoid t_REAL s in T and the bounds a, b: the result is then guaranteed and we use a fast algorithm (Uspensky's method, relying on Descartes's rule of sign, see polrootsreal). Otherwise, the polynomial is rescaled and rounded first and the result may be wrong due to that initial error. If only a or b is inexact, on the other hand, the interval is first thickened using rational endpoints and the result remains guaranteed unless there exist a root *very* close to a nonrational endpoint (which may be missed or unduly included).

```
? T = (x-1)*(x-2)*(x-3);
? polsturm(T)
%2 = 3
? polsturm(T, [-oo,2])
%3 = 2
? polsturm(T, [1/2,+oo])
%4 = 3
? polsturm(T, [1, Pi]) \\ Pi inexact: not recommended !
%5 = 3
? polsturm(T*1., [0, 4]) \\ T*1. inexact: not recommended !
%6 = 3
? polsturm(T*2, [0, 4]) \\ not squarefree: roots are not repeated!
%7 = 3
```

polsylvestermatrix(y)

Forms the Sylvester matrix corresponding to the two polynomials x and y, where the coefficients of the polynomials are put in the columns of the matrix (which is the natural direction for solving equations afterwards). The use of this matrix can be essential when dealing with polynomials with inexact entries, since polynomial Euclidean division doesn't make much sense in this case.

polsym(n)

Creates the column vector of the symmetric powers of the roots of the polynomial x up to power n, using Newton's formula.

polteichmuller(p, r)

Given $T \in \mathbb{F}_p[X]$ return the polynomial $P \in \mathbb{Z}_p[X]$ whose roots (resp. leading coefficient) are the Teichmuller lifts of the roots (resp. leading coefficient) of T, to p-adic precision r. If T is monic, P is the reduction modulo p^r of the unique monic polynomial congruent to T modulo p such that $P(X^p) = 0(modP(X), p^r)$.

```
? T = ffinit(3, 3, 't)
%1 = Mod(1,3)*t^3 + Mod(1,3)*t^2 + Mod(1,3)*t + Mod(2,3)
? P = polteichmuller(T,3,5)
%2 = t^3 + 166*t^2 + 52*t + 242
? subst(P, t, t^3) % (P*Mod(1,3^5))
%3 = Mod(0, 243)
? [algdep(a+0(3^5),2) | a <- Vec(P)]
%4 = [x - 1, 5*x^2 + 1, x^2 + 4*x + 4, x + 1]
```

When T is monic and irreducible mod p, this provides a model $\mathbb{Q}_p[X]/(P)$ of the unramified extension $\mathbb{Q}_p[X]/(T)$ where the Frobenius has the simple form $XmodP: --> X^pmodP$.

poltschirnhaus()

Applies a random Tschirnhausen transformation to the polynomial x, which is assumed to be nonconstant and separable, so as to obtain a new equation for the étale algebra defined by x. This is for instance useful when computing resolvents, hence is used by the **polgalois** function.

polylogmult(z, t, precision)

For s a vector of positive integers and z a vector of complex numbers of the same length, returns the multiple polylogarithm value (MPV)

$$\zeta(s_1, ..., s_r; z_1, ..., z_r) = \sum_{n_1 > ... > n_r > 0} \prod_{1 < i < r} z_i^{n_i} / n_i^{s_i}.$$

If z is omitted, assume z = [1, ..., 1], i.e., Multiple Zeta Value. More generally, return Yamamoto's interpolation between ordinary multiple polylogarithms (t = 0) and star polylogarithms (t = 1, using the condition $n_1 > =$... $> = n_r > 0$), evaluated at t.

We must have $||z_1...z_i|| \le 1$ for all i, and if $s_1 = 1$ we must have $z_1! = 1$.

? 8*polylogmult([2,1],[-1,1]) - zeta(3) %1 = 0.E-38

Warning. The algorithm used converges when the z_i are 1. It may not converge as some $z_i! = 1$ becomes too close to 1, even at roots of 1 of moderate order:

```
? polylogmult([2,1], (99+20*I)/101 * [1,1])
   *** polylogmult: sorry, polylogmult in this range is not yet implemented.
? polylogmult([2,1], exp(I*Pi/20)* [1,1])
   *** polylogmult: sorry, polylogmult in this range is not yet implemented.
```

More precisely, if $y_i := 1/(z_1...z_i)$ and

$$v := \min_{i < j; y_i! = 1} \| (1 - y_i) y_j \| > 1/4$$

then the algorithm computes the value up to a 2^{-b} absolute error in $O(k^2N)$ operations on floating point numbers of O(N) bits, where $k = \sum_i s_i$ is the weight and $N = b/\log_2(4v)$.

powers(n, x0)

For nonnegative n, return the vector with n+1 components $[1, x, ..., x^n]$ if **x0** is omitted, and $[x_0, x_0 * x, ..., x_0 * x^n]$ otherwise.

```
? powers(Mod(3,17), 4)
%1 = [Mod(1, 17), Mod(3, 17), Mod(9, 17), Mod(10, 17), Mod(13, 17)]
? powers(Mat([1,2;3,4]), 3)
```

```
%2 = [[1, 0; 0, 1], [1, 2; 3, 4], [7, 10; 15, 22], [37, 54; 81, 118]]
? powers(3, 5, 2)
%3 = [2, 6, 18, 54, 162, 486]
```

When n < 0, the function returns the empty vector [].

precision(n)

The function behaves differently according to whether n is present or not. If n is missing, the function returns the floating point precision in decimal digits of the PARI object x. If x has no floating point component, the function returns +00.

```
? precision(exp(1e-100))
%1 = 154 \\ 154 significant decimal digits
? precision(2 + x)
%2 = +oo \\ exact object
? precision(0.5 + 0(x))
%3 = 38 \\ floating point accuracy, NOT series precision
? precision( [ exp(1e-100), 0.5 ] )
%4 = 38 \\ minimal accuracy among components
```

Using getlocalprec() allows to retrieve the working precision (as modified by possible localprec statements).

If *n* is present, the function creates a new object equal to *x* with a new floating point precision *n*: *n* is the number of desired significant *decimal* digits. If *n* is smaller than the precision of a t_REAL component of *x*, it is truncated, otherwise it is extended with zeros. For non-floating-point types, no change.

precprime()

Finds the largest pseudoprime (see ispseudoprime) less than or equal to x. x can be of any real type. Returns 0 if $x \le 1$. Note that if x is a prime, this function returns x and not the largest prime strictly smaller than x. To rigorously prove that the result is prime, use isprime.

primecert(flag)

If N is a prime, return a PARI Primality Certificate for the prime N, as described below. Otherwise, return 0. A Primality Certificate c can be checked using primecertisvalid(c).

If flag = 0 (default), return an ECPP certificate (Atkin-Morain)

A PARI ECPP Primality Certificate for the prime N is either a prime integer $N < 2^{64}$ or a vector C of length ℓ whose i is a vector $[N_i, t_i, s_i, a_i, P_i]$ of length 5 where $N_1 = N$. It is said to be *valid* if for each $i = 1, ..., \ell$, all of the following conditions are satisfied

- N_i is a positive integer
- t_i is an integer such that $t_i^2 < 4N_i$
- s_i is a positive integer which divides m_i where $m_i = N_i + 1 t_i$
- If we set $q_i = (m_i)/(s_i)$, then

*
$$q_i > (N_i^{1/4} + 1)^2$$

* $q_i = N_{i+1}$ if 1 <= i < l

* $q_\ell <= 2^{64}$ is prime

• a_i is an integer

* P[i] is a vector of length 2 representing the affine point $P_i = (x_i, y_i)$ on the elliptic curve $E : y^2 = x^3 + a_i x + b_i$ modulo N_i where $b_i = y_i^2 - x_i^3 - a_i x_i$ satisfying the following:

* $m_i P_i = oo$

 $* s_i P_i! = oo$

Using the following theorem, the data in the vector C allows to recursively certify the primality of N (and all the q_i) under the single assumption that q_ℓ be prime.

Theorem. If N is an integer and there exist positive integers m, q and a point P on the elliptic curve E: $y^2 = x^3 + ax + b$ defined modulo N such that $q > (N^{1/4} + 1)^2$, q is a prime divisor of m, mP = oo and (m)/(q)P! = oo, then N is prime.

If flag = 1 (very slow), return an N - 1 certificate (Pocklington Lehmer)

A PARI N - 1 Primality Certificate for the prime N is either a prime integer $N < 2^{64}$ or a pair [N, C], where C is a vector with ℓ elements which are either a single integer $p_i < 2^{64}$ or a triple $[p_i, a_i, C_i]$ with $p_i > 2^{64}$ satisfying the following properties:

- p_i is a prime divisor of N-1;
- a_i is an integer such that $a_i^{N-1} = 1 \pmod{N}$ and $a_i^{(N-1)/p_i} 1$ is coprime with N;
- C_i is an N-1 Primality Certificate for p_i
- The product F of the $p_i^{v_{p_i}(N-1)}$ is strictly larger than $N^{1/3}$. Provided that all p_i are indeed primes, this implies that any divisor of N is congruent to 1 modulo F.
- The Billhart, Lehmer, Selfridge criterion is satisfied: when we write $N = 1 + c_1F + c_2F^2$ in base F the polynomial $1 + c_1X + c_2X^2$ is irreducible over \mathbb{Z} , i.e. $c_1^2 4c_2$ is not a square. This implies that N is prime.

This algorithm requires factoring partially p - 1 for various prime integers p with an unfactored parted $\langle = p^{2/3}$ and this may be exceedingly slow compared to the default.

The algorithm fails if one of the pseudo-prime factors is not prime, which is exceedingly unlikely and well worth a bug report. Note that if you monitor the algorithm at a high enough debug level, you may see warnings about untested integers being declared primes. This is normal: we ask for partial factorizations (sufficient to prove primality if the unfactored part is not too large), and factor warns us that the cofactor hasn't been tested. It may or may not be tested later, and may or may not be prime. This does not affect the validity of the whole Primality Certificate.

primecertexport(format)

Returns a string suitable for print/write to display a primality certificate from primecert, the format of which depends on the value of format:

- 0 (default): Human-readable format. See ??primecert for the meaning of the successive N, t, s, a, m, q, E, P. The integer D is the negative fundamental discriminant coredisc $(t^2 4N)$.
- 1: Primo format 4.
- 2: MAGMA format.

Currently, only ECPP Primality Certificates are supported.

```
? cert = primecert(10^35+69);
? s = primecertexport(cert); \\ Human-readable
? print(s)
[1]
t = 546867911035452074
s = 2963504668391148
a = 0
D = -3
m = 999999999999999999453132088964547996
q = 33743830764501150277
E = [0, 1]
P = [21567861682493263464353543707814204],
49167839501923147849639425291163552]
۲<u>2</u>1
N = 33743830764501150277
t = -11610830419
s = 734208843
a = 0
D = -3
m = 33743830776111980697
q = 45959444779
E = [0, 25895956964997806805]
P = [29257172487394218479, 3678591960085668324]
\\ Primo format
? s = primecertexport(cert,1); write("cert.out", s);
\\ Magma format, write to file
? s = primecertexport(cert,2); write("cert.m", s);
? cert = primecert(10^{35+69}, 1); \\ N-1 certificate
? primecertexport(cert)
 *** at top-level: primecertexport(cert)
        _____
 *** primecertexport: sorry, N-1 certificate is not yet implemented.
```

primecertisvalid()

Verifies if cert is a valid PARI ECPP Primality certificate, as described in ??primecert.

```
, [18022351516, 9326882 51]]]
? primecertisvalid(cert)
%2 = 1
? cert[1][1]++; \\ random perturbation
? primecertisvalid(cert)
%4 = 0 \\ no longer valid
? primecertisvalid(primecert(6))
%5 = 0
```

primepi()

The prime counting function. Returns the number of primes $p, p \le x$.

```
? primepi(10)
%1 = 4;
? primes(5)
%2 = [2, 3, 5, 7, 11]
? primepi(10^11)
%3 = 4118054813
```

Uses checkpointing and a naive O(x) algorithm; make sure to start gp with primelimit at least \sqrt{x} .

primes()

Creates a row vector whose components are the first n prime numbers. (Returns the empty vector for $n \le 0$.) A t_VEC n = [a, b] is also allowed, in which case the primes in [a, b] are returned

```
? primes(10) \\ the first 10 primes
%1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([0,29]) \\ the primes up to 29
%2 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes([15,30])
%3 = [17, 19, 23, 29]
```

prodeulerrat(s, a, precision)

 $\prod_{p>=a} F(p^s)$, where the product is taken over prime numbers and F is a rational function.

```
? prodeulerrat(1+1/q^3,1)
%1 = 1.1815649490102569125693997341604542605
? zeta(3)/zeta(6)
%2 = 1.1815649490102569125693997341604542606
```

prodnumrat(a, precision)

 $\prod_{n>=a} F(n)$, where F-1 is a rational function of degree less than or equal to -2.

```
? prodnumrat(1+1/x^2,1)
%1 = 3.6760779103749777206956974920282606665
```

psdraw(flag)

This function is obsolete, use plotexport and write the result to file.

psi(precision)

The ψ -function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

psplothraw(listy, flag)

This function is obsolete, use plothrawexport and write the result to file.

qfauto(fl)

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the automorphism group of the associate lattice. Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension grows. G can also be given by an qfisominit structure. See qfisominit for the meaning of fl.

The output is a two-components vector [o, g] where o is the group order and g is the list of generators (as a vector). For each generator H, the equality $G = {}^{t}HGH$ holds.

The interface of this function is experimental and will likely change in the future.

This function implements an algorithm of Plesken and Souvignier, following Souvignier's implementation.

qfautoexport(flag)

qfa being an automorphism group as output by qfauto, export the underlying matrix group as a string suitable for (no flags or flag = 0) GAP or (flag = 1) Magma. The following example computes the size of the matrix group using GAP:

```
? G = qfauto([2,1;1,2])
%1 = [12, [[-1, 0; 0, -1], [0, -1; 1, 1], [1, 1; 0, -1]]]
? s = qfautoexport(G)
%2 = "Group([[-1, 0], [0, -1]], [[0, -1], [1, 1]], [[1, 1], [0, -1]])"
? extern("echo \"Order("s");\" | gap -q")
%3 = 12
```

qfbclassno(flag)

Ordinary class number of the quadratic order of discriminant D, for "small" values of D.

- if D > 0 or flag = 1, use a $O(||D||^{1/2})$ algorithm (compute $L(1, \chi_D)$ with the approximate functional equation). This is slower than quadclassunit as soon as $||D|| 10^2$ or so and is not meant to be used for large D.
- if D < 0 and flag = 0 (or omitted), use a $O(||D||^{1/4})$ algorithm (Shanks's baby-step/giant-step method). It should be faster than quadclassunit for small values of D, say $||D|| < 10^{18}$.

Important warning. In the latter case, this function only implements part of Shanks's method (which allows to speed it up considerably). It gives unconditionnally correct results for $||D|| < 2.10^{10}$, but may give incorrect results for larger values if the class group has many cyclic factors. We thus recommend to double-check results using the function quadclassunit, which is about 2 to 3 times slower in the range $||D|| \in [10^{10}, 10^{18}]$, assuming GRH. We currently have no counter-examples but they should exist: we would appreciate a bug report if you find one.

Warning. Contrary to what its name implies, this routine does not compute the number of classes of binary primitive forms of discriminant D, which is equal to the *narrow* class number. The two notions are the same when D < 0 or the fundamental unit ε has negative norm; when D > 0 and $N\varepsilon > 0$, the number of classes of forms is twice the ordinary class number. This is a problem which we cannot fix for backward compatibility reasons. Use the following routine if you are only interested in the number of classes of forms:

Here are a few examples:

```
? qfbclassno(400000028) \\ D > 0: slow
time = 3,140 ms.
%1 = 1
? quadclassunit(400000028).no
time = 20 ms. \\{ much faster, assume GRH}
%2 = 1
? qfbclassno(-400000028) \\ D < 0: fast enough
time = 0 ms.
%3 = 7253
? quadclassunit(-400000028).no
time = 0 ms.
%4 = 7253
```

See also qfbhclassno.

qfbcompraw(y)

composition of the binary quadratic forms x and y, without reduction of the result. This is useful e.g. to compute a generating element of an ideal. The result is undefined if x and y do not have the same discriminant.

qfbhclassno()

Hurwitz class number of x, when x is nonnegative and congruent to 0 or 3 modulo 4, and 0 for other values. For $x > 5.10^5$, we assume the GRH, and use quadclassunit with default parameters.

```
? qfbhclassno(1) \\ not 0 or 3 mod 4
%1 = 0
? qfbhclassno(3)
%2 = 1/3
? qfbhclassno(4)
%3 = 1/2
? qfbhclassno(23)
%4 = 3
```

qfbil(*y*, *q*)

This function is obsolete, use qfeval.

qfbnucomp(y, L)

composition of the primitive positive definite binary quadratic forms x and y (type t_QFI) using the NUCOMP and NUDUPL algorithms of Shanks, à la Atkin. L is any positive constant, but for optimal speed, one should take $L = ||D/4||^{1/4}$, i.e. sqrtnint(abs(D) >> 2,4), where D is the common discriminant of x and y. When x and y do not have the same discriminant, the result is undefined.

The current implementation is slower than the generic routine for small D, and becomes faster when D has about 45 bits.

qfbnupow(n, L)

n-th power of the primitive positive definite binary quadratic form x using Shanks's NUCOMP and NUDUPL algorithms; if set, L should be equal to sqrtnint(abs(D) >> 2,4), where D < 0 is the discriminant of x.

The current implementation is slower than the generic routine for small discriminant D, and becomes faster for $D 2^{45}$.

qfbpowraw(n)

n-th power of the binary quadratic form x, computed without doing any reduction (i.e. using qfbcompraw). Here n must be nonnegative and $n < 2^{31}$.

qfbprimeform(p, precision)

Prime binary quadratic form of discriminant x whose first coefficient is p, where ||p|| is a prime number. By abuse of notation, p = 1 is also valid and returns the unit form. Returns an error if x is not a quadratic residue mod p, or if x < 0 and p < 0. (Negative definite t_QFI are not implemented.) In the case where x > 0, the "distance" component of the form is set equal to zero according to the current precision.

qfbred(flag, d, isd, sd)

Reduces the binary quadratic form x (updating Shanks's distance function if x is indefinite). The binary digits of flag are toggles meaning

1: perform a single reduction step

2: don't update Shanks's distance

The arguments d, *isd*, *sd*, if present, supply the values of the discriminant, $floor\sqrt{d}$, and \sqrt{d} respectively (no checking is done of these facts). If d < 0 these values are useless, and all references to Shanks's distance are irrelevant.

qfbredsl2(data)

Reduction of the (real or imaginary) binary quadratic form x, return [y, g] where y is reduced and g in $SL(2, \mathbb{Z})$ is such that g.x = y; *data*, if present, must be equal to [D, sqrtint(D)], where D > 0 is the discriminant of x. In case x is a t_QFR, the distance component is unaffected.

qfbsolve(n, flag)

Solve the equation Q(x, y) = n in coprime integers x and y (primitive solutions), where Q is a binary quadratic form and n an integer, up to the action of the special orthogonal group $G = SO(Q, \mathbb{Z})$, which is isomorphic to the group of units of positive norm of the quadratic order of discriminant D = discQ. If D > 0, G is infinite. If D < -4, G is of order 2, if D = -3, G is of order 6 and if D = -4, G is of order 4.

Binary digits of flag mean: 1: return all solutions if set, else a single solution; return [] if a single solution is wanted (bit unset) but none exist. 2: also include imprimitive solutions.

When flag = 2 (return a single solution, possibly imprimitive), the algorithm returns a solution with minimal content; in particular, a primitive solution exists if and only if one is returned.

The integer *n* can be given by its factorization matrix :emphasis: fa = factor(n) or by the pair [n, fa].

```
? qfbsolve(Qfb(1,0,2), 603) \\ a single primitive solution
%1 = [5, 17]
? qfbsolve(Qfb(1,0,2), 603, 1) \\ all primitive solutions
%2 = [[5, 17], [-19, -11], [19, -11], [5, -17]]
? qfbsolve(Qfb(1,0,2), 603, 2) \\ a single, possibly imprimitive solution
%3 = [5, 17] \\ actually primitive
? qfbsolve(Qfb(1,0,2), 603, 3) \\ all solutions
%4 = [[5, 17], [-19, -11], [19, -11], [5, -17], [-21, 9], [-21, -9]]
? N = 2^128+1; F = factor(N);
? qfbsolve(Qfb(1,0,1), [N,F],1)
%3 = [[-16382350221535464479,8479443857936402504],
[18446744073709551616,-1], [-18446744073709551616,-1],
[16382350221535464479,8479443857936402504]]
```

For fixed Q, assuming the factorisation of n is given, the algorithm runs in probabilistic polynomial time in $\log p$, where p is the largest prime divisor of n, through the computation of square roots of D modulo 4p). The dependency on Q is more complicated: polynomial time in $\log |D||$ if Q is imaginary, but exponential time if Q

is real (through the computation of a full cycle of reduced forms). In the latter case, note that bnfisprincipal provides a solution in heuristic subexponential time assuming the GRH.

qfeval(x, y)

Evaluate the quadratic form q (given by a symmetric matrix) at the vector x; if y is present, evaluate the polar form at (x, y); if q omitted, use the standard Euclidean scalar product, corresponding to the identity matrix.

Roughly equivalent to $x \sim * q * y$, but a little faster and more convenient (does not distinguish between column and row vectors):

? x = [1,2,3]~; y = [-1,3,1]~; q = [1,2,3;2,2,-1;3,-1,9]; ? qfeval(q,x,y) %2 = 23 ? for(i=1,10^6, qfeval(q,x,y)) time = 661ms ? for(i=1,10^6, x~*q*y) time = 697ms

The speedup is noticeable for the quadratic form, compared to $x \sim * q * x$, since we save almost half the operations:

? for(i=1,10^6, qfeval(q,x))
time = 487ms

The special case q = Id is handled faster if we omit q altogether:

? qfeval(,x,y) %6 = 8 ? q = matid(#x); ? for(i=1,10^6, qfeval(q,x,y)) time = 529 ms. ? for(i=1,10^6, qfeval(,x,y)) time = 228 ms. ? for(i=1,10^6, x~*y) time = 274 ms.

We also allow t_MAT s of compatible dimensions for x, and return $\mathbf{x} \sim \mathbf{x} \mathbf{q} \mathbf{x}$ in this case as well:

```
? M = [1,2,3;4,5,6;7,8,9]; qfeval(,M) \\ Gram matrix
%5 =
[66 78 90]
[78 93 108]
[90 108 126]
? q = [1,2,3;2,2,-1;3,-1,9];
? for(i=1,10^6, qfeval(q,M))
time = 2,008 ms.
? for(i=1,10^6, M~*q*M)
time = 2,368 ms.
? for(i=1,10^6, qfeval(,M))
time = 1,053 ms.
```

? for(i=1,10⁶, M~*M) time = 1,171 ms.

If q is a t_QFI or t_QFR, it is implicitly converted to the attached symmetric t_MAT. This is done more efficiently than by direct conversion, since we avoid introducing a denominator 2 and rational arithmetic:

```
? q = Qfb(2,3,4); x = [2,3];
? qfeval(q, x)
%2 = 62
? Q = Mat(q)
%3 =
  [ 2 3/2]
[3/2 4]
? qfeval(Q, x)
%4 = 62
? for (i=1, 10^6, qfeval(q,x))
time = 758 ms.
? for (i=1, 10^6, qfeval(Q,x))
time = 1,110 ms.
```

Finally, when x is a t_MAT with *integral* coefficients, we allow a t_QFI or t_QFR for q and return the binary quadratic form qoM. Again, the conversion to t_MAT is less efficient in this case:

```
? q = Qfb(2,3,4); Q = Mat(q); x = [1,2;3,4];
? qfeval(q, x)
%2 = Qfb(47, 134, 96)
? qfeval(Q,x)
%3 =
[47 67]
[67 96]
? for (i=1, 10^6, qfeval(q,x))
time = 701 ms.
? for (i=1, 10^6, qfeval(Q,x))
time = 1,639 ms.
```

qfgaussred()

decomposition into squares of the quadratic form represented by the symmetric matrix q. The result is a matrix whose diagonal entries are the coefficients of the squares, and the off-diagonal entries on each line represent the bilinear forms. More precisely, if (a_{ij}) denotes the output, one has

$$q(x) = \sum_{i} a_{ii} (x_i + \sum_{j!=i} a_{ij} x_j)^2$$

? qfgaussred([0,1;1,0]) %1 = [1/2 1] [-1 -1/2]

This means that $2xy = (1/2)(x + y)^2 - (1/2)(x - y)^2$. Singular matrices are supported, in which case some diagonal coefficients will vanish:

? qfgaussred([1,1;1,1]) %1 = [1 1]

[1 0]

This means that $x^2 + 2xy + y^2 = (x + y)^2$.

qfisom(H, fl, grp)

G, H being square and symmetric matrices with integer entries representing positive definite quadratic forms, return an invertible matrix S such that $G = {}^{t}SHS$. This defines a isomorphism between the corresponding lattices. Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension grows. See qfisominit for the meaning of *fl*. If *grp* is given it must be the automorphism group of H. It will be used to speed up the computation.

G can also be given by an qfisominit structure which is preferable if several forms H need to be compared to G.

This function implements an algorithm of Plesken and Souvignier, following Souvignier's implementation.

qfisominit(fl, m)

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, return an isom structure allowing to compute isomorphisms between G and other quadratic forms faster.

The interface of this function is experimental and will likely change in future release.

If present, the optional parameter fl must be a t_VEC with two components. It allows to specify the invariants used, which can make the computation faster or slower. The components are

- fl[1] Depth of scalar product combination to use.
- f1[2] Maximum level of Bacher polynomials to use.

If present, m must be the set of vectors of norm up to the maximal of the diagonal entry of G, either as a matrix or as given by qfminim. Otherwise this function computes the minimal vectors so it become very lengthy as the dimension of G grows.

qfjacobi(precision)

Apply Jacobi's eigenvalue algorithm to the real symmetric matrix A. This returns [L, V], where

- L is the vector of (real) eigenvalues of A, sorted in increasing order,
- V is the corresponding orthogonal matrix of eigenvectors of A.

```
? norml2( (A-L[1])*H[,1] ) \\ approximate eigenvector
%5 = 9.403954806578300064 E-38
? norml2(H*H~ - 1)
%6 = 2.350988701644575016 E-38 \\ close to orthogonal
```

qflll(flag)

LLL algorithm applied to the *columns* of the matrix x. The columns of x may be linearly dependent. The result is by default a unimodular transformation matrix T such that x.T is an LLL-reduced basis of the lattice generated by the column vectors of x. Note that if x is not of maximal rank T will not be square. The LLL parameters are (0.51, 0.99), meaning that the Gram-Schmidt coefficients for the final basis satisfy $\|\mu_{i,j}\| \le 0.51$, and the Lovász's constant is 0.99.

If flag = 0 (default), assume that x has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in flag = 1.

If flag = 1, assume that x is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (fplll-1.3).

If flag = 2, x should be an integer matrix whose columns are linearly independent. Returns a partially reduced basis for x, using an unpublished algorithm by Peter Montgomery: a basis is said to be *partially reduced* if $||v_iv_j|| \ge ||v_i||$ for any two distinct basis vectors v_i, v_j . This is faster than flag = 1, esp. when one row is huge compared to the other rows (knapsack-style), and should quickly produce relatively short vectors. The resulting basis is *not* LLL-reduced in general. If LLL reduction is eventually desired, avoid this partial reduction: applying LLL to the partially reduced matrix is significantly *slower* than starting from a knapsack-type lattice.

If flag = 3, as flag = 1, but the reduction is performed in place: the routine returns x.T. This is usually faster for knapsack-type lattices.

If flag = 4, as flag = 1, returning a vector [K, T] of matrices: the columns of K represent a basis of the integer kernel of x (not LLL-reduced in general) and T is the transformation matrix such that x.T is an LLL-reduced \mathbb{Z} -basis of the image of the matrix x.

If flag = 5, case as case 4, but x may have polynomial coefficients.

If flag = 8, same as case 0, but x may have polynomial coefficients.

```
? \p500
realprecision = 500 significant digits
? a = 2*cos(2*Pi/97);
? C = 10^{450};
? v = powers(a,48); b = round(matconcat([matid(48),C*v]~));
? p = b * qflll(b)[,1]; \\ tiny linear combination of powers of 'a'
time = 4,470 ms.
? exponent(v * p / C)
\%5 = -1418
? p3 = qflll(b,3)[,1]; \setminus compute in place, faster
time = 3,790 ms.
? p3 == p \ same result
%7 = 1
? p2 = b * qflll(b,2)[,1]; \\ partial reduction: faster, not as good
time = 343 \text{ ms}.
? exponent(v * p2 / C)
\%9 = -1190
```

qflllgram(flag)

Same as qflll, except that the matrix G = x * x is the Gram matrix of some lattice vectors x, and not the coordinates of the vectors themselves. In particular, G must now be a square symmetric real matrix, corresponding to a positive quadratic form (not necessarily definite: x needs not have maximal rank). The result is a unimodular transformation matrix T such that x.T is an LLL-reduced basis of the lattice generated by the column vectors of x. See qflll for further details about the LLL implementation.

If flag = 0 (default), assume that G has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in flag = 1.

If flag = 1, assume that G is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (fplll-1.3).

flag = 4: G has integer entries, gives the kernel and reduced image of x.

flag = 5: same as 4, but G may have polynomial coefficients.

qfminim(*B*, *m*, *flag*, *precision*)

x being a square and symmetric matrix of dimension d representing a positive definite quadratic form, this function deals with the vectors of x whose norm is less than or equal to B, enumerated using the Fincke-Pohst algorithm, storing at most m pairs of vectors: only one vector is given for each pair v. There is no limit if m is omitted: beware that this may be a huge vector! The vectors are returned in no particular order.

The function searches for the minimal nonzero vectors if B is omitted. The behavior is undefined if x is not positive definite (a "precision too low" error is most likely, although more precise error messages are possible). The precise behavior depends on flag.

- If flag = 0 (default), return [N, M, V], where N is the number of vectors enumerated (an even number, possibly larger than 2m), $M \ll B$ is the maximum norm found, and V is a matrix whose columns are found vectors.
- If flag = 1, ignore m and return [M, v], where v is a nonzero vector of length $M \le B$. If no nonzero vector has length $\le B$, return []. If no explicit B is provided, return a vector of smallish norm, namely the vector of smallest length (usually the first one but not always) in an LLL-reduced basis for x.

In these two cases, x must have integral *small* entries: more precisely, we definitely must have $d \cdot ||x||_o o^2 < 2^{53}$ but even that may not be enough. The implementation uses low precision floating point computations for maximal speed and gives incorrect results when x has large entries. That condition is checked in the code and the routine raises an error if large rounding errors occur. A more robust, but much slower, implementation is chosen if the following flag is used:

• If flag = 2, x can have non integral real entries, but this is also useful when x has large integral entries. Return [N, M, V] as in case flag = 0, where M is returned as a floating point number. If x is inexact and B is omitted, the "minimal" vectors in V only have approximately the same norm (up to the internal working accuracy). This version is very robust but still offers no hard and fast guarantee about the result: it involves floating point operations performed at a high floating point precision depending on your input, but done without rigorous tracking of roundoff errors (as would be provided by interval arithmetic for instance). No example is known where the input is exact but the function returns a wrong result.

```
? x = matid(2);
? qfminim(x) \setminus 4 minimal vectors of norm 1: \pm[0,1], \pm[1,0]
%2 = [4, 1, [0, 1; 1, 0]]
? { x = \setminus The Leech lattice
[4, 2, 0, 0, 0, -2, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, -2;
2, 4, -2, -2, 0, -2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, -1, 0, 1, -1, -1;
0, -2, 4, 0, -2, 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 0, 0, 1, -1, -1, 0, 0;
0, -2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 1, -1, 1, 0;
```

```
0, 0, -2, 0, 4, 0, 0, 0, 1, -1, 0, 0, 1, 0, 0, 0, -2, 0, 0, -1, 1, 1, 0,
                                                                         0;
-2, -2, 0, 0, 0, 4, -2, 0, -1, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, -1, 1, 1;
0, 0, 0, 0, 0, -2, 4, -2, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, 1, -1, 0;
0, 0, 0, 0, 0, 0, -2, 4, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, -1, -1, 0, 1, 0;
0, 0, 0, 0, 1,-1, 0, 0, 4, 0,-2, 0, 1, 1, 0,-1, 0, 1, 0, 0, 0, 0, 0, 0,
                                                                         0;
0, 0, 0, 0, -1, 0, 0, 0, 0, 4, 0, 0, 1, 1, -1, 1, 0, 0, 0, 1, 0, 0, 1,
                                                                          0:
0, 0, 0, 0, 0, 0, 0, 0, -2, 0, 4, -2, 0, -1, 0, 0, 0, -1, 0, -1, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2, 4, -1, 1, 0, 0, -1, 1, 0, 1, 1, 1, -1, 0;
1, 0, -1, 1, 1, 0, 0, -1, 1, 1, 0, -1, 4, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, -1;
-1, -1, 1, -1, 0, 0, 1, 0, 1, 1, -1, 1, 0, 4, 1, 1, 0, 0, 1, 1, 0, 1, 0,
                                                                         1;
0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 1, 4, 0, 0, 0, 1, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 1, 0, 4, 0, 0, 0, 0, 1, 1, 0, 0;
0, 0, 1, 0, -2, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 4, 1, 1, 1, 0, 0, 1, 1;
1, 0, 0, 1, 0, 0, -1, 0, 1, 0, -1, 1, 1, 0, 0, 0, 1, 4, 0, 1, 1, 0, 1,
                                                                         0;
0, 0, 0, -1, 0, 1, 0, -1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 4, 0, 1, 1, 0, 1;
-1, -1, 1, 0, -1, 1, 0, -1, 0, 1, -1, 1, 0, 1, 0, 0, 1, 1, 0, 4, 0, 0, 1, 1;
0, 0, -1, 1, 1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 4, 1, 0, 1;
0, 1,-1,-1, 1,-1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 4, 0, 1;
0, -1, 0, 1, 0, 1, -1, 1, 0, 1, 0, -1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 4, 1;
-2, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 4]; \}
? qfminim(x,,0) \setminus 0: don't store minimal vectors
time = 121 \text{ ms.}
%4 = [196560, 4, [;]] \\ 196560 minimal vectors of norm 4
? qfminim(x) \\ store all minimal vectors !
time = 821 \text{ ms.}
? qfminim(x, 0, 2); \\ safe algorithm. Slower and unnecessary here.
time = 5.540 ms.
\%6 = [196560, 4.000061035156250000, [;]]
? qfminim(x,,,2); \\ safe algorithm; store all minimal vectors
time = 6,602 ms.
```

In this example, storing 0 vectors limits memory use; storing all of them requires a parisize about 50MB. All minimal vectors are nevertheless enumerated in both cases of course, which means the speedup is likely to be marginal.

qfnorm(q)

This function is obsolete, use qfeval.

qforbits(V)

Return the orbits of V under the action of the group of linear transformation generated by the set G. It is assumed that G contains minus identity, and only one vector in v, -v should be given. If G does not stabilize V, the function return 0.

In the example below, we compute representatives and lengths of the orbits of the vectors of norm ≤ 3 under the automorphisms of the lattice \mathbb{Z}^6 .

```
? Q=matid(6); G=qfauto(Q); V=qfminim(Q,3);
? apply(x->[x[1],#x],qforbits(G,V))
%2 = [[[0,0,0,0,0,0,1]~,6],[[0,0,0,0,1,-1]~,30],[[0,0,0,1,-1,-1]~,80]]
```

qfparam(sol, flag)

Coefficients of binary quadratic forms that parametrize the solutions of the ternary quadratic form G, using the particular solution *sol. flag* is optional and can be 1, 2, or 3, in which case the *flag*-th form is reduced. The default is *flag* = 0 (no reduction).

? G = [1,0,0;0,1,0;0,0,-34]; ? M = qfparam(G, qfsolve(G)) %2 = [3 -10 -3] [-5 -6 5] [1 0 1]

Indeed, the solutions can be parametrized as

$$(3x^{2} - 10xy - 3y^{2})^{2} + (-5x^{2} - 6xy + 5y^{2})^{2} - 34(x^{2} + y^{2})^{2} = 0.$$

? $v = y^2 * M^{1}[1, x/y, (x/y)^{2}]$ ~ %3 = $[3^{x}x^{2} - 10^{y}x - 3^{y}y^{2}, -5^{x}x^{2} - 6^{y}x + 5^{y}y^{2}, -x^{2} - y^{2}]$ ~ ? $v^{G^{x}v}$ %4 = 0

qfperfection()

G being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the perfection rank of the form. That is, gives the rank of the family of the s symmetric matrices vv^t , where v runs through the minimal vectors.

The algorithm computes the minimal vectors and its runtime is exponential in the dimension of x.

qfrep(B, flag)

q being a square and symmetric matrix with integer entries representing a positive definite quadratic form, count the vectors representing successive integers.

- If flag = 0, count all vectors. Outputs the vector whose *i*-th entry, $1 \le i \le B$ is half the number of vectors v such that q(v) = i.
- If flag = 1, count vectors of even norm. Outputs the vector whose *i*-th entry, $1 \le i \le B$ is half the number of vectors such that q(v) = 2i.

```
? q = [2, 1; 1, 3];
? qfrep(q, 5)
%2 = Vecsmall([0, 1, 2, 0, 0]) \\ 1 vector of norm 2, 2 of norm 3, etc.
? qfrep(q, 5, 1)
%3 = Vecsmall([1, 0, 0, 1, 0]) \\ 1 vector of norm 2, 0 of norm 4, etc.
```

This routine uses a naive algorithm based on qfminim, and will fail if any entry becomes larger than 2^{31} (or 2^{63}).

qfsign()

Returns [p, m] the signature of the quadratic form represented by the symmetric matrix x. Namely, p (resp. m) is the number of positive (resp. negative) eigenvalues of x. The result is computed using Gaussian reduction.

qfsolve()

Given a square symmetric matrix G of dimension $n \ge 1$, solve over \mathbb{Q} the quadratic equation $X^t G X = 0$. The matrix G must have rational coefficients. The solution might be a single nonzero vector (vectorv) or a matrix (whose columns generate a totally isotropic subspace).

If no solution exists, returns an integer, that can be a prime p such that there is no local solution at p, or -1 if there is no real solution, or -2 if n = 2 and $-\det G$ is positive but not a square (which implies there is a real solution, but no local solution at some p dividing det G).

```
? G = [1,0,0;0,1,0;0,0,-34];
? qfsolve(G)
%1 = [-3, -5, 1]~
? qfsolve([1,0; 0,2])
%2 = -1 \\ no real solution
? qfsolve([1,0,0;0,3,0; 0,0,-2])
%3 = 3 \\ no solution in Q_3
? qfsolve([1,0; 0,-2])
%4 = -2 \\ no solution, n = 2
```

quadclassunit(flag, tech, precision)

Buchmann-McCurley's sub-exponential algorithm for computing the class group of a quadratic order of discriminant D.

This function should be used instead of qfbclassno or quadregulator when $D < -10^{25}$, $D > 10^{10}$, or when the *structure* is wanted. It is a special case of bnfinit, which is slower, but more robust.

The result is a vector v whose components should be accessed using member functions:

- :math:`v.no`: the class number
- :math:`v.cyc`: a vector giving the structure of the class group as a product of cyclic groups;
- :math:`v.gen`: a vector giving generators of those cyclic groups (as binary quadratic forms).
- :math:`v.reg`: the regulator, computed to an accuracy which is the maximum of an internal accuracy determined by the program and the current default (note that once the regulator is known to a small accuracy it is trivial to compute it to very high accuracy, see the tutorial).

The flag is obsolete and should be left alone. In older versions, it supposedly computed the narrow class group when D > 0, but this did not work at all; use the general function bnfnarrow.

Optional parameter *tech* is a row vector of the form $[c_1, c_2]$, where $c_1 <= c_2$ are nonnegative real numbers which control the execution time and the stack size, see GRHbnf (in the PARI manual). The parameter is used as a threshold to balance the relation finding phase against the final linear algebra. Increasing the default c_1 means that relations are easier to find, but more relations are needed and the linear algebra will be harder. The default value for c_1 is 0 and means that it is taken equal to c_2 . The parameter c_2 is mostly obsolete and should not be changed, but we still document it for completeness: we compute a tentative class group by generators and relations using a factorbase of prime ideals $\langle = c_1(\log ||D||)^2$, then prove that ideals of norm $\langle = c_2(\log ||D||)^2$ do not generate a larger group. By default an optimal c_2 is chosen, so that the result is provably correct under the GRH — a famous result of Bach states that $c_2 = 6$ is fine, but it is possible to improve on this algorithmically. You may provide a smaller c_2 , it will be ignored (we use the provably correct one); you may provide a larger c_2 than the default value, which results in longer computing times for equally correct outputs (under GRH).

quaddisc()

Discriminant of the étale algebra $\mathbb{Q}(\sqrt{x})$, where $x \in \mathbb{Q}^*$. This is the same as $\operatorname{coredisc}(d)$ where d is the integer squarefree part of x, so $x = df^2$ with $f \in \mathbb{Q}^*$ and $d \in \mathbb{Z}$. This returns 0 for x = 0, 1 for x square and the discriminant of the quadratic field $\mathbb{Q}(\sqrt{x})$ otherwise.

? quaddisc(7) %1 = 28 ? quaddisc(-7) %2 = -7

quadgen(v)

Creates the quadratic number $\omega = (a + \sqrt{D})/2$ where a = 0 if $D = 0 \mod 4$, a = 1 if $D = 1 \mod 4$, so that $(1, \omega)$
is an integral basis for the quadratic order of discriminant D. D must be an integer congruent to 0 or 1 modulo 4, which is not a square. If v is given, the variable name is used to display g else 'w' is used.

```
? w = quadgen(5, 'w); w^2 - w - 1
%1 = 0
? w = quadgen(0, 'w)
*** at top-level: w=quadgen(0)
*** ^____
*** quadgen: domain error in quadpoly: issquare(disc) = 1
```

quadhilbert(precision)

Relative equation defining the Hilbert class field of the quadratic field of discriminant D.

If D < 0, uses complex multiplication (Schertz's variant).

If D > 0 Stark units are used and (in rare cases) a vector of extensions may be returned whose compositum is the requested class field. See bnrstark for details.

quadpoly(v)

Creates the "canonical" quadratic polynomial (in the variable v) corresponding to the discriminant D, i.e. the minimal polynomial of quadgen(D). D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

```
? quadpoly(5,'y)
%1 = y^2 - y - 1
? quadpoly(0,'y)
*** at top-level: quadpoly(0,'y)
*** ^------
*** quadpoly: domain error in quadpoly: issquare(disc) = 1
```

quadray(f, precision)

Relative equation for the ray class field of conductor f for the quadratic field of discriminant D using analytic methods. A bnf for $x^2 - D$ is also accepted in place of D.

For D < 0, uses the σ function and Schertz's method.

For D > 0, uses Stark's conjecture, and a vector of relative equations may be returned. See bnrstark for more details.

quadregulator(precision)

Regulator of the quadratic field of positive discriminant x. Returns an error if x is not a discriminant (fundamental or not) or if x is a square. See also quadclassunit if x is large.

quadunit(v)

Fundamental unit u of the real quadratic field $\mathbb{Q}(\sqrt{D})$ where D is the positive discriminant of the field. If D is not a fundamental discriminant, this probably gives the fundamental unit of the corresponding order. D must be an integer congruent to 0 or 1 modulo 4, which is not a square; the result is a quadratic number (see quadgen (in the PARI manual)). If v is given, the variable name is used to display u else 'w' is used. The algorithm computes the continued fraction of $(1+\sqrt{D})/2$ or $\sqrt{D}/2$ (see GTM 138, algorithm 5.7.2). Although the continued fraction length is only $O(\sqrt{D})$, the function still runs in time O(D), in part because the output size is not polynomially bounded in terms of log D. See bnfinit and bnfunits for a better alternative for large D, running in time subexponential in log D and returning the fundamental units in compact form (as a short list of S-units of size $O(\log D)^3$ raised to possibly large exponents).

ramanujantau()

Compute the value of Ramanujan's tau function at an individual n, assuming the truth of the GRH (to compute quickly class numbers of imaginary quadratic fields using quadclassunit). Algorithm in $O(n^{1/2})$ using

 $O(\log n)$ space. If all values up to N are required, then

$$\sum \tau(n)q^{n} = q \prod_{n>=1} (1-q^{n})^{24}$$

will produce them in time O(N), against $O(N^{3/2})$ for individual calls to ramanujantau; of course the space complexity then becomes O(N).

```
? tauvec(N) = Vec(q*eta(q + 0(q^N))^24);
? N = 10^4; v = tauvec(N);
time = 26 ms.
? ramanujantau(N)
%3 = -482606811957501440000
? w = vector(N, n, ramanujantau(n)); \\ much slower !
time = 13,190 ms.
? v == w
%4 = 1
```

random()

Returns a random element in various natural sets depending on the argument N.

- t_INT: returns an integer uniformly distributed between 0 and N 1. Omitting the argument is equivalent to random(2^31).
- t_REAL: returns a real number in [0, 1[with the same accuracy as N (whose mantissa has the same number of significant words).
- t_INTMOD: returns a random intmod for the same modulus.
- t_FFELT: returns a random element in the same finite field.
- t_VEC of length 2, N = [a, b]: returns an integer uniformly distributed between a and b.
- t_VEC generated by ellinit over a finite field k (coefficients are t_INTMOD s modulo a prime or t_FFELT s): returns a "random" k-rational affine point on the curve. More precisely if the curve has a single point (at infinity!) we return it; otherwise we return an affine point by drawing an abscissa uniformly at random until ellordinate succeeds. Note that this is definitely not a uniform distribution over E(k), but it should be good enough for applications.
- t_POL return a random polynomial of degree at most the degree of N. The coefficients are drawn by applying random to the leading coefficient of N.

```
? random(10)
%1 = 9
? random(Mod(0,7))
%2 = Mod(1, 7)
? a = ffgen(ffinit(3,7), 'a); random(a)
%3 = a^6 + 2*a^5 + a^4 + a^3 + a^2 + 2*a
? E = ellinit([3,7]*Mod(1,109)); random(E)
%4 = [Mod(103, 109), Mod(10, 109)]
? E = ellinit([1,7]*a^0); random(E)
%5 = [a^6 + a^5 + 2*a^4 + 2*a^2, 2*a^6 + 2*a^4 + 2*a^3 + a^2 + 2*a]
? random(Mod(1,7)*x^4)
%6 = Mod(5, 7)*x^4 + Mod(6, 7)*x^3 + Mod(2, 7)*x^2 + Mod(2, 7)*x + Mod(5, 7)
```

These variants all depend on a single internal generator, and are independent from your operating system's random number generators. A random seed may be obtained via getrand, and reset using setrand: from a given seed, and given sequence of random s, the exact same values will be generated. The same seed is used at each startup,

reseed the generator yourself if this is a problem. Note that internal functions also call the random number generator; adding such a function call in the middle of your code will change the numbers produced.

Technical note. Up to version 2.4 included, the internal generator produced pseudo-random numbers by means of linear congruences, which were not well distributed in arithmetic progressions. We now use Brent's XORGEN algorithm, based on Feedback Shift Registers, see http://wwwmaths.anu.edu.au/~brent/random.html. The generator has period $2^{4096} - 1$, passes the Crush battery of statistical tests of L'Ecuyer and Simard, but is not suitable for cryptographic purposes: one can reconstruct the state vector from a small sample of consecutive values, thus predicting the entire sequence.

randomprime(q)

Returns a strong pseudo prime (see ispseudoprime) in [2, N - 1]. A t_VEC N = [a, b] is also allowed, with $a \le b$ in which case a pseudo prime $a \le p \le b$ is returned; if no prime exists in the interval, the function will run into an infinite loop. If the upper bound is less than 2^{64} the pseudo prime returned is a proven prime.

If the optional parameter q is an integer, return a prime congruent to 1modq; if q is an intmod, return a prime in the given congruence class. If the class contains no prime in the given interval, the function will raise an exception if the class is not invertible, else run into an infinite loop

real()

Real part of x. When x is a quadratic number, this is the coefficient of 1 in the "canonical" integral basis $(1, \omega)$.

? real(3 + I)
%1 = 3
? x = 3 + quadgen(-23);
? real(x) \\ as a quadratic number
%3 = 3

removeprimes()

Removes the primes listed in x from the prime number table. In particular removeprimes(addprimes()) empties the extra prime table. x can also be a single integer. List the current extra primes if x is omitted.

rnfalgtobasis(x)

Expresses x on the relative integral basis. Here, rnf is a relative number field extension L/K as output by rnfinit, and x an element of L in absolute form, i.e. expressed as a polynomial or polmod with polmod coefficients, *not* on the relative integral basis.

rnfbasis(M)

Let K the field represented by *bnf*, as output by bnfinit. M is a projective \mathbb{Z}_K -module of rank n ($M \otimes K$ is an *n*-dimensional K-vector space), given by a pseudo-basis of size n. The routine returns either a true \mathbb{Z}_K -basis of M (of size n) if it exists, or an n + 1-element generating set of M if not.

It is allowed to use a monic irreducible polynomial P in K[X] instead of M, in which case, M is defined as the ring of integers of K[X]/(P), viewed as a \mathbb{Z}_K -module.

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes an order which is maximal at all maximal ideals specified by B, see ??rnfinit: the valuation of D is then correct at all such maximal ideals but may be incorrect at other primes.

rnfbasistoalg(*x*)

Computes the representation of x as a polmod with polmods coefficients. Here, rnf is a relative number field extension L/K as output by rnfinit, and x an element of L expressed on the relative integral basis.

rnfcharpoly(T, a, var)

Characteristic polynomial of a over nf, where a belongs to the algebra defined by T over nf, i.e. nf[X]/(T). Returns a polynomial in variable v (x by default).

? nf = nfinit(y²+1); ? rnfcharpoly(nf, x²+y*x+1, x+y) %2 = x² + Mod(-y, y² + 1)*x + 1

rnfconductor(*T*, *flag*)

Given a *bnf* structure attached to a number field K, as produced by bnfinit, and T an irreducible polynomial in K[x] defining an Abelian extension L = K[x]/(T), computes the class field theory conductor of this Abelian extension. If T does not define an Abelian extension over K, the result is undefined; it may be the integer 0 (in which case the extension is definitely not Abelian) or a wrong result.

The result is a 3-component vector [f, bnr, H], where f is the conductor of the extension given as a 2-component row vector $[f_0, f_o o]$, *bnr* is the attached **bnr** structure and H is a matrix in HNF defining the subgroup of the ray class group on the ray class group generators **bnr**.gen; in particular, it is a left divisor of the diagonal matrix attached to **bnr**.cyc and $\|\det H\| = N = \deg T$.

If flag is set, return [f, bnrmod, H], where bnrmod is now attached to Cl_f/Cl_f^N , and H is as before since it contains the N-th powers. This is useful when f contains a maximal ideal with huge residue field, since the corresponding tough discrete logarithms are trivialized: in the quotient group, all elements have small order dividing N. This allows to work in Cl_f/H but no longer in Cl_f .

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes the conductor of the extension provided it factors completely over the maximal ideals specified by B, see ??rnfinit: the valuation of f_0 is then correct at all such maximal ideals but may be incorrect at other primes.

rnfdedekind(pol, pr, flag)

Given a number field K coded by nf and a monic polynomial $P \in \mathbb{Z}_K[X]$, irreducible over K and thus defining a relative extension L of K, applies Dedekind's criterion to the order $\mathbb{Z}_K[X]/(P)$, at the prime ideal pr. It is possible to set pr to a vector of prime ideals (test maximality at all primes in the vector), or to omit altogether, in which case maximality at all primes is tested; in this situation flag is automatically set to 1.

The default historic behavior (*flag* is 0 or omitted and *pr* is a single prime ideal) is not so useful since **rnfpseudobasis** gives more information and is generally not that much slower. It returns a 3-component vector [max, basis, v]:

- *basis* is a pseudo-basis of an enlarged order O produced by Dedekind's criterion, containing the original order $\mathbb{Z}_K[X]/(P)$ with index a power of pr. Possibly equal to the original order.
- *max* is a flag equal to 1 if the enlarged order O could be proven to be *pr*-maximal and to 0 otherwise; it may still be maximal in the latter case if *pr* is ramified in L,
- v is the valuation at pr of the order discriminant.

If *flag* is nonzero, on the other hand, we just return 1 if the order $\mathbb{Z}_K[X]/(P)$ is *pr*-maximal (resp. maximal at all relevant primes, as described above), and 0 if not. This is much faster than the default, since the enlarged order is not computed.

```
? nf = nfinit(y<sup>2</sup>-3); P = x<sup>3</sup> - 2*y;
? pr3 = idealprimedec(nf,3)[1];
? rnfdedekind(nf, P, pr3)
%3 = [1, [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, 1]], 8]
? rnfdedekind(nf, P, pr3, 1)
%4 = 1
```

In this example, pr3 is the ramified ideal above 3, and the order generated by the cube roots of y is already pr3-maximal. The order-discriminant has valuation 8. On the other hand, the order is not maximal at the prime above 2:

? pr2 = idealprimedec(nf,2)[1]; ? rnfdedekind(nf, P, pr2, 1) %6 = 0 ? rnfdedekind(nf, P, pr2) %7 = [0, [[2, 0, 0; 0, 1, 0; 0, 0, 1], [[1, 0; 0, 1], [1, 0; 0, 1], [1, 1/2; 0, 1/2]]], 2]

The enlarged order is not proven to be pr2-maximal yet. In fact, it is; it is in fact the maximal order:

```
? B = rnfpseudobasis(nf, P)
%8 = [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, [1, 1/2; 0, 1/2]],
[162, 0; 0, 162], -1]
? idealval(nf,B[3], pr2)
%9 = 2
```

It is possible to use this routine with nonmonic $P = \sum_{i \le n} p_i X^i \in \mathbb{Z}_K[X]$ if flag = 1; in this case, we test maximality of Dedekind's order generated by

$$1, p_n \alpha, p_n \alpha^2 + p_{n-1} \alpha, \dots, p_n \alpha^{n-1} + p_{n-1} \alpha^{n-2} + \dots + p_1 \alpha$$

The routine will fail if P vanishes on the projective line over the residue field \mathbb{Z}_K/pr (FIXME).

rnfdet(M)

Given a pseudo-matrix M over the maximal order of nf, computes its determinant.

rnfdisc(T)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial $T \in K[x]$ defining a relative extension L = K[x]/(T), compute the relative discriminant of L. This is a vector [D, d], where D is the relative ideal discriminant and d is the relative discriminant considered as an element of K^*/K^{*2} . The main variable of *nf must* be of lower priority than that of T, see priority (in the PARI manual).

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and computes an order which is maximal at all maximal ideals specified by B, see ??rnfinit: the valuation of D is then correct at all such maximal ideals but may be incorrect at other primes.

rnfeltabstorel(x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an element of L expressed as a polynomial modulo the absolute equation :emphasis:`rnf.pol`, or in terms of the absolute \mathbb{Z} -basis for \mathbb{Z}_L if rnf contains one (as in rnfinit(nf,pol,1), or after a call to nfinit(rnf)). Computes x as an element of the relative extension L/K as a polmod with polmod coefficients.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? L.polabs
\%2 = x^{4} + 1
? rnfeltabstorel(L, Mod(x, L.polabs))
\%3 = Mod(x, x^2 + Mod(-y, y^2 + 1))
? rnfeltabstorel(L, 1/3)
\%4 = 1/3
? rnfeltabstorel(L, Mod(x, x^2-y))
\%5 = Mod(x, x^2 + Mod(-y, y^2 + 1))
? rnfeltabstorel(L, [0,0,0,1]~) \\ Z_L not initialized yet
 *** at top-level: rnfeltabstorel(L,[0,
 *** A_____
 *** rnfeltabstorel: incorrect type in rnfeltabstorel, apply nfinit(rnf).
? nfinit(L); \\ initialize now
? rnfeltabstorel(L, [0,0,0,1]~)
\%6 = Mod(Mod(y, y^2 + 1)*x, x^2 + Mod(-y, y^2 + 1))
```

rnfeltdown(x, flag)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L expressed as a polynomial or polmod with polmod coefficients (or as a t_COL on nfinit(rnf).zk), computes x as an element of K as a t_POLMOD if flag = 0 and as a t_COL otherwise. If x is not in K, a domain error occurs.

```
? rnfeltdown(L, Mod(y, x^2-y), 1) \\ as a t_COL
%7 = [0, 1]~
? rnfeltdown(L, [0,1,0,0]~) \\ not allowed without absolute nf struct
 *** rnfeltdown: incorrect type in rnfeltdown (t_COL).
? nfinit(L); \\ add absolute nf structure to L
? rnfeltdown(L, [0,1,0,0]~) \\ now OK
%8 = Mod(y, y^2 + 1)
```

If we had started with $L = rnfinit(K, x^2-y, 1)$, then the final would have worked directly.

rnfeltnorm(x)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L, returns the relative norm $N_{L/K}(x)$ as an element of K.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? rnfeltnorm(L, Mod(x, L.pol))
%2 = Mod(x, x^2 + Mod(-y, y^2 + 1))
? rnfeltnorm(L, 2)
%3 = 4
? rnfeltnorm(L, Mod(x, x^2-y))
```

rnfeltreltoabs(x)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of the absolute extension L/\mathbb{Q} as a polynomial modulo the absolute equation :emphasis: rnf.pol.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? L.pol
%2 = x^4 + 1
? rnfeltreltoabs(L, Mod(x, L.pol))
%3 = Mod(x, x^4 + 1)
? rnfeltreltoabs(L, Mod(y, x^2-y))
%4 = Mod(x^2, x^4 + 1)
? rnfeltreltoabs(L, Mod(y,K.pol))
%5 = Mod(x^2, x^4 + 1)
```

rnfelttrace(*x*)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of L, returns the relative trace $Tr_{L/K}(x)$ as an element of K.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y);
? rnfelttrace(L, Mod(x, L.pol))
%2 = 0
? rnfelttrace(L, 2)
%3 = 4
? rnfelttrace(L, Mod(x, x^2-y))
```

rnfeltup(x, flag)

rnf being a relative number field extension L/K as output by rnfinit and x being an element of K, computes x as an element of the absolute extension L/\mathbb{Q} . As a t_POLMOD modulo :emphasis:`rnf.pol` if flag = 0 and as a t_COL on the absolute field integer basis if flag = 1.

```
? K = nfinit(y<sup>2</sup>+1); L = rnfinit(K, x<sup>2</sup>-y);
? L.pol
%2 = x<sup>4</sup> + 1
? rnfeltup(L, Mod(y, K.pol))
%3 = Mod(x<sup>2</sup>, x<sup>4</sup> + 1)
? rnfeltup(L, y)
%4 = Mod(x<sup>2</sup>, x<sup>4</sup> + 1)
? rnfeltup(L, [1,2]~) \\ in terms of K.zk
%5 = Mod(2*x<sup>2</sup> + 1, x<sup>4</sup> + 1)
? rnfeltup(L, y, 1) \\ in terms of nfinit(L).zk
%6 = [0, 1, 0, 0]~
? rnfeltup(L, [1,2]~, 1)
%7 = [1, 2, 0, 0]~
```

rnfequation(pol, flag)

Given a number field nf as output by nfinit (or simply a polynomial) and a polynomial *pol* with coefficients in nf defining a relative extension L of nf, computes an absolute equation of L over \mathbb{Q} .

The main variable of nf must be of lower priority than that of pol (see priority (in the PARI manual)). Note that for efficiency, this does not check whether the relative equation is irreducible over nf, but only if it is squarefree. If it is reducible but squarefree, the result will be the absolute equation of the étale algebra defined by pol. If pol is not squarefree, raise an e_DOMAIN exception.

```
? rnfequation(y<sup>2</sup>+1, x<sup>2</sup> - y)
%1 = x<sup>4</sup> + 1
? T = y<sup>3</sup>-2; rnfequation(nfinit(T), (x<sup>3</sup>-2)/(x-Mod(y,T)))
%2 = x<sup>6</sup> + 108 \\ Galois closure of Q(2<sup>(1/3)</sup>)
```

If flag is nonzero, outputs a 3-component row vector [z, a, k], where

- z is the absolute equation of L over \mathbb{Q} , as in the default behavior,
- a expresses as a t_POLMOD modulo z a root α of the polynomial defining the base field nf,
- k is a small integer such that θ = β + kα is a root of z, where β is a root of pol. It is guaranteed that k = 0 whenever Q(β) = L.

```
? T = y^3-2; pol = x^2 +x*y + y^2;
? [z,a,k] = rnfequation(T, pol, 1);
? z
%3 = x^6 + 108
? subst(T, y, a)
%4 = 0
? alpha= Mod(y, T);
? beta = Mod(x*Mod(1,T), pol);
? subst(z, x, beta + k*alpha)
%7 = 0
```

rnfhnfbasis(x)

Given bnf as output by bnfinit, and either a polynomial x with coefficients in bnf defining a relative extension L of bnf, or a pseudo-basis x of such an extension, gives either a true bnf-basis of L in upper triangular Hermite normal form, if it exists, and returns 0 otherwise.

rnfidealabstorel(x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an ideal of the absolute

extension L/\mathbb{Q} . Returns the relative pseudo-matrix in HNF giving the ideal x considered as an ideal of the relative extension L/K, i.e. as a \mathbb{Z}_K -module.

Let Labs be an (absolute) nf structure attached to L, obtained via Labs = nfinit(rnf)). Then rnf "knows" about Labs and x may be given in any format attached to Labs, e.g. a prime ideal or an ideal in HNF wrt. Labs.zk:

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y); Labs = nfinit(rnf);
? m = idealhnf(Labs, 17, x^3+2); \\ some ideal in HNF wrt. Labs.zk
? B = rnfidealabstorel(rnf, m)
%3 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]] \\ pseudo-basis for m as Z_K-module
? A = rnfidealreltoabs(rnf, B)
%4 = [17, x^2 + 4, x + 8, x^3 + 8*x^2] \\ Z-basis for m in Q[x]/(rnf.polabs)
? mathnf(matalgtobasis(Labs, A)) == m
%5 = 1
```

If on the other hand, we do not have a Labs at hand, because it would be too expensive to compute, but we nevertheless have a \mathbb{Z} -basis for x, then we can use the function with this basis as argument. The entries of x may be given either modulo rnf.polabs (absolute form, possibly lifted) or modulo rnf.pol (relative form as t_POLMOD s):

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y);
? rnfidealabstorel(rnf, [17, x^2 + 4, x + 8, x^3 + 8*x^2])
%2 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]]
? rnfidealabstorel(rnf, Mod([17, y + 4, x + 8, y*x + 8*y], x^2-y))
%3 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]]
```

rnfidealdown(x)

Let rnf be a relative number field extension L/K as output by rnfinit, and x an ideal of L, given either in relative form or by a \mathbb{Z} -basis of elements of L (see rnfidealabstorel (in the PARI manual)). This function returns the ideal of K below x, i.e. the intersection of x with K.

rnfidealfactor(x)

Factor into prime ideal powers the ideal x in the attached absolute number field L = nfinit(rnf). The output format is similar to the factor function, and the prime ideals are represented in the form output by the idealprimedec function for L.

```
? rnf = rnfinit(nfinit(y^2+1), x^2-y+1);
? rnfidealfactor(rnf, y+1) \\ P_2^2
%2 =
[[2, [0,0,1,0]~, 4, 1, [0,0,0,2;0,0,-2,0;-1,-1,0,0;1,-1,0,0]] 2]
? rnfidealfactor(rnf, x) \\ P_2
%3 =
[[2, [0,0,1,0]~, 4, 1, [0,0,0,2;0,0,-2,0;-1,-1,0,0;1,-1,0,0]] 1]
? L = nfinit(rnf);
? id = idealhnf(L, idealhnf(L, 25, (x+1)^2));
? idealfactor(L, id) == rnfidealfactor(rnf, id)
%6 = 1
```

Note that ideals of the base field K must be explicitly lifted to L via rnfidealup before they can be factored.

rnfidealhnf(*x*)

rnf being a relative number field extension L/K as output by **rnfinit** and x being a relative ideal (which can be,

as in the absolute case, of many different types, including of course elements), computes the HNF pseudo-matrix attached to x, viewed as a \mathbb{Z}_K -module.

rnfidealmul(x, y)

rnf being a relative number field extension L/K as output by rnfinit and x and y being ideals of the relative extension L/K given by pseudo-matrices, outputs the ideal product, again as a relative ideal.

rnfidealnormabs(x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal (which can be, as in the absolute case, of many different types, including of course elements). This function computes the norm of the x considered as an ideal of the absolute extension L/\mathbb{Q} . This is identical to

```
idealnorm(rnf, rnfidealnormrel(rnf,x))
```

but faster.

rnfidealnormrel(x)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal (which can be, as in the absolute case, of many different types, including of course elements). This function computes the relative norm of x as an ideal of K in HNF.

rnfidealprimedec(pr)

Let *rnf* be a relative number field extension L/K as output by rnfinit, and *pr* a maximal ideal of *K* (*prid*), this function completes the *rnf* with a *nf* structure attached to *L* (see rnfinit (in the PARI manual)) and returns the prime ideal decomposition of *pr* in L/K.

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^3+y+1);
? P = idealprimedec(K, 2)[1];
? S = rnfidealprimedec(rnf, P);
? #S
%4 = 1
```

The argument pr is also allowed to be a prime number p, in which case the function returns a pair of vectors [SK,SL], where SK contains the primes of K above p and SL[i] is the vector of primes of L above SK[i].

```
? [SK,SL] = rnfidealprimedec(rnf, 5);
? [#SK, vector(#SL,i,#SL[i])]
%6 = [2, [2, 2]]
```

rnfidealreltoabs(x, flag)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be a relative ideal, given as a \mathbb{Z}_K -module by a pseudo matrix [A, I]. This function returns the ideal x as an absolute ideal of L/\mathbb{Q} . If flag = 0, the result is given by a vector of t_POLMOD s modulo rnf.pol forming a \mathbb{Z} -basis; if flag = 1, it is given in HNF in terms of the fixed \mathbb{Z} -basis for \mathbb{Z}_L , see rnfinit (in the PARI manual).

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y);
? P = idealprimedec(K,2)[1];
? P = rnfidealup(rnf, P)
%3 = [2, x^2 + 1, 2*x, x^3 + x]
? Prel = rnfidealhnf(rnf, P)
%4 = [[1, 0; 0, 1], [[2, 1; 0, 1], [2, 1; 0, 1]]]
? rnfidealreltoabs(rnf,Prel)
%5 = [2, x^2 + 1, 2*x, x^3 + x]
? rnfidealreltoabs(rnf,Prel,1)
%6 =
```

[2 1 0 0] [0 1 0 0] [0 0 2 1] [0 0 0 1]

The reason why we do not return by default (flag = 0) the customary HNF in terms of a fixed \mathbb{Z} -basis for \mathbb{Z}_L is precisely because a *rnf* does not contain such a basis by default. Completing the structure so that it contains a *nf* structure for *L* is polynomial time but costly when the absolute degree is large, thus it is not done by default. Note that setting flag = 1 will complete the *rnf*.

rnfidealtwoelt(x)

rnf being a relative number field extension L/K as output by rnfinit and x being an ideal of the relative extension L/K given by a pseudo-matrix, gives a vector of two generators of x over \mathbb{Z}_L expressed as polmods with polmod coefficients.

rnfidealup(x, flag)

Let rnf be a relative number field extension L/K as output by rnfinit and let x be an ideal of K. This function returns the ideal $x\mathbb{Z}_L$ as an absolute ideal of L/\mathbb{Q} , in the form of a \mathbb{Z} -basis. If flag = 0, the result is given by a vector of polynomials (modulo rnf.pol); if flag = 1, it is given in HNF in terms of the fixed \mathbb{Z} -basis for \mathbb{Z}_L , see rnfinit (in the PARI manual).

```
? K = nfinit(y^2+1); rnf = rnfinit(K, x^2-y);
? P = idealprimedec(K,2)[1];
? rnfidealup(rnf, P)
%3 = [2, x^2 + 1, 2*x, x^3 + x]
? rnfidealup(rnf, P,1)
%4 =
[2 1 0 0]
[0 1 0 0]
[0 0 2 1]
[0 0 0 1]
```

The reason why we do not return by default (flag = 0) the customary HNF in terms of a fixed \mathbb{Z} -basis for \mathbb{Z}_L is precisely because a *rnf* does not contain such a basis by default. Completing the structure so that it contains a *nf* structure for *L* is polynomial time but costly when the absolute degree is large, thus it is not done by default. Note that setting flag = 1 will complete the *rnf*.

rnfinit(T, flag)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial T in $\mathbb{Z}_K[x]$ defining a relative extension L = K[x]/(T), this computes data to work in L/K The main variable of T must be of higher priority (see **priority** (in the PARI manual)) than that of nf, and the coefficients of T must be in K.

The result is a row vector, whose components are technical. We let $m = [K : \mathbb{Q}]$ the degree of the base field, n = [L : K] the relative degree, r_1 and r_2 the number of real and complex places of K. Access to this information via *member functions* is preferred since the specific data organization specified below will change in the future.

If flag = 1, add an *nf* structure attached to *L* to *rnf*. This is likely to be very expensive if the absolute degree mn is large, but fixes an integer basis for \mathbb{Z}_L as a \mathbb{Z} -module and allows to input and output elements of *L* in

absolute form: as t_COL for elements, as t_MAT in HNF for ideals, as prid for prime ideals. Without such a call, elements of L are represented as t_POLMOD , etc. Note that a subsequent nfinit(rnf) will also explicitly add such a component, and so will the following functions rnfidealmul, rnfidealtwoelt, rnfidealprimedec, rnfidealup (with flag 1) and rnfidealreltoabs (with flag 1). The absolute *nf* structure attached to L can be recovered using nfinit(rnf).

rnf[1]) contains the relative polynomial T.

rnf[2] contains the integer basis [A, d] of K, as (integral) elements of L/\mathbb{Q} . More precisely, A is a vector of polynomial with integer coefficients, d is a denominator, and the integer basis is given by A/d.

rnf[3] (rnf.disc) is a two-component row vector [d(L/K), s] where d(L/K) is the relative ideal discriminant of L/K and s is the discriminant of L/K viewed as an element of $K^*/(K^*)^2$, in other words it is the output of rnfdisc.

rnf[4]) is the ideal index f, i.e. such that $d(T)\mathbb{Z}_K = f^2 d(L/K)$.

rnf[5]) is the list of rational primes dividing the norm of the relative discriminant ideal.

rnf[7] (rnf.zk) is the pseudo-basis (A, I) for the maximal order \mathbb{Z}_L as a \mathbb{Z}_K -module: A is the relative integral pseudo basis expressed as polynomials (in the variable of T) with polmod coefficients in nf, and the second component I is the ideal list of the pseudobasis in HNF.

rnf[8] is the inverse matrix of the integral basis matrix, with coefficients polmods in nf.

rnf[9] is currently unused.

rnf[10] (rnf.nf) is nf.

rnf[11] is an extension of rnfequation(K, T, 1). Namely, a vector [P, a, k, K.pol, T] describing the *absolute* extension L/\mathbb{Q} : P is an absolute equation, more conveniently obtained as rnf.polabs; a expresses the generator $\alpha = ymodK.pol$ of the number field K as an element of L, i.e. a polynomial modulo the absolute equation P;

k is a small integer such that, if β is an abstract root of T and α the generator of K given above, then $P(\beta + k\alpha) = 0$. It is guaranteed that k = 0 if $\mathbb{Q}(\beta) = L$.

Caveat. Be careful if k! = 0 when dealing simultaneously with absolute and relative quantities since $L = \mathbb{Q}(\beta + k\alpha) = K(\alpha)$, and the generator chosen for the absolute extension is not the same as for the relative one. If this happens, one can of course go on working, but we advise to change the relative polynomial so that its root becomes $\beta + k\alpha$. Typical GP instructions would be

```
[P,a,k] = rnfequation(K, T, 1);
if (k, T = subst(T, x, x - k*Mod(y, K.pol)));
L = rnfinit(K, T);
```

rnf[12] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available (which is rarely needed, hence would be too expensive to compute during the initial rnfinit call).

Huge discriminants, helping rnfdisc. When T has a discriminant which is difficult to factor, it is hard to compute \mathbb{Z}_L . As in nfinit, the special input format [T, B] is also accepted, where T is a polynomial as above and B specifies a list of maximal ideals. The following formats are recognized for B:

- an integer: the list of all maximal ideals above a rational prime p < B.
- a vector of rational primes or prime ideals: the list of all maximal ideals dividing an element in the list.

Instead of \mathbb{Z}_L , this produces an order which is maximal at all such maximal ideals primes. The result may actually be a complete and correct *rnf* structure if the relative ideal discriminant factors completely over this list of maximal ideals but this is not guaranteed. In general, the order may not be maximal at primes p not in the list such that p^2 divides the relative ideal discriminant.

rnfisabelian(T)

T being a relative polynomial with coefficients in nf, return 1 if it defines an abelian extension, and 0 otherwise.

```
? K = nfinit(y<sup>2</sup> + 23);
? rnfisabelian(K, x<sup>3</sup> - 3*x - y)
%2 = 1
```

rnfisfree(*x*)

Given bnf as output by bnfinit, and either a polynomial x with coefficients in bnf defining a relative extension L of bnf, or a pseudo-basis x of such an extension, returns true (1) if L/bnf is free, false (0) if not.

rnfislocalcyclo()

Let *rnf* be a relative number field extension L/K as output by **rnfinit** whose degree [L : K] is a power of a prime ℓ . Return 1 if the ℓ -extension is locally cyclotomic (locally contained in the cyclotomic \mathbb{Z}_{ℓ} -extension of K_v at all places $v \parallel \ell$), and 0 if not.

```
? K = nfinit(y<sup>2</sup> + y + 1);
? L = rnfinit(K, x<sup>3</sup> - y); /* = K(zeta_9), globally cyclotomic */
? rnfislocalcyclo(L)
%3 = 1
\\ we expect 3-adic continuity by Krasner's lemma
? vector(5, i, rnfislocalcyclo(rnfinit(K, x<sup>3</sup> - y + 3<sup>i</sup>)))
%5 = [0, 1, 1, 1, 1]
```

rnfisnorm(a, flag)

Similar to bnfisnorm but in the relative case. T is as output by rnfisnorminit applied to the extension L/K. This tries to decide whether the element a in K is the norm of some x in the extension L/K.

The output is a vector [x, q], where a = Norm(x) * q. The algorithm looks for a solution x which is an S-integer, with S a list of places of K containing at least the ramified primes, the generators of the class group of L, as well as those primes dividing a. If L/K is Galois, then this is enough but you may want to add more primes to S to produce different elements, possibly smaller; otherwise, flag is used to add more primes to S: all the places above the primes p <= flag (resp. p || flag) if flag > 0 (resp. flag < 0).

The answer is guaranteed (i.e. *a* is a norm iff q = 1) if the field is Galois, or, under GRH, if *S* contains all primes less than $12 \log^2 ||\operatorname{disc}(M)||$, where *M* is the normal closure of L/K.

If rnfisnorminit has determined (or was told) that L/K is Galois, and flag! = 0, a Warning is issued (so that you can set flag = 1 to check whether L/K is known to be Galois, according to T). Example:

```
bnf = bnfinit(y<sup>3</sup> + y<sup>2</sup> - 2*y - 1);
p = x<sup>2</sup> + Mod(y<sup>2</sup> + 2*y + 1, bnf.pol);
T = rnfisnorminit(bnf, p);
rnfisnorm(T, 17)
```

checks whether 17 is a norm in the Galois extension $\mathbb{Q}(\beta)/\mathbb{Q}(\alpha)$, where $\alpha^3 + \alpha^2 - 2\alpha - 1 = 0$ and $\beta^2 + \alpha^2 + 2\alpha + 1 = 0$ (it is).

rnfisnorminit(polrel, flag)

Let K be defined by a root of *pol*, and L/K the extension defined by the polynomial *polrel*. As usual, *pol* can in fact be an *nf*, or *bnf*, etc; if *pol* has degree 1 (the base field is \mathbb{Q}), polrel is also allowed to be an *nf*, etc. Computes technical data needed by **rnfisnorm** to solve norm equations Nx = a, for x in L, and a in K.

If flag = 0, do not care whether L/K is Galois or not.

If flag = 1, L/K is assumed to be Galois (unchecked), which speeds up rnfisnorm.

If flag = 2, let the routine determine whether L/K is Galois.

rnfkummer(subgp, precision)

This function is deprecated, use bnrclassfield.

rnflllgram(pol, order, precision)

Given a polynomial *pol* with coefficients in *nf* defining a relative extension L and a suborder *order* of L (of maximal rank), as output by rnfpseudobasis(nf, pol) or similar, gives [[neworder], U], where *neworder* is a reduced order and U is the unimodular transformation matrix.

rnfnormgroup(pol)

bnr being a big ray class field as output by brinit and *pol* a relative polynomial defining an Abelian extension, computes the norm group (alias Artin or Takagi group) corresponding to the Abelian extension of bnf = bnr. bnf defined by *pol*, where the module corresponding to *bnr* is assumed to be a multiple of the conductor (i.e. *pol* defines a subextension of bnr). The result is the HNF defining the norm group on the given generators of bnr. gen. Note that neither the fact that *pol* defines an Abelian extension nor the fact that the module is a multiple of the conductor is checked. The result is undefined if the assumption is not correct, but the function will return the empty matrix [;] if it detects a problem; it may also not detect the problem and return a wrong result.

rnfpolred(pol, precision)

This function is obsolete: use rnfpolredbest instead. Relative version of polred. Given a monic polynomial *pol* with coefficients in nf, finds a list of relative polynomials defining some subfields, hopefully simpler and containing the original field. In the present version **2.13.3**, this is slower and less efficient than rnfpolredbest.

Remark. This function is based on an incomplete reduction theory of lattices over number fields, implemented by rnflllgram, which deserves to be improved.

rnfpolredabs(pol, flag)

Relative version of polredabs. Given an irreducible monic polynomial pol with coefficients in the maximal order of nf, finds a canonical relative polynomial defining the same field, hopefully with small coefficients. Note that the equation is only canonical for a fixed nf, using a different defining polynomial in the nf structure will produce a different relative equation.

The binary digits of flag correspond to 1: add information to convert elements to the new representation, 2: absolute polynomial, instead of relative, 16: possibly use a suborder of the maximal order. More precisely:

0: default, return P

1: returns [P, a] where P is the default output and a, a t_POLMOD modulo P, is a root of pol.

2: returns *Pabs*, an absolute, instead of a relative, polynomial. This polynomial is canonical and does not depend on the *nf* structure. Same as but faster than

polredabs(rnfequation(nf, pol))

3: returns [Pabs, a, b], where *Pabs* is an absolute polynomial as above, a, b are t_POLMOD modulo *Pabs*, roots of nf.pol and *pol* respectively.

16: (OBSOLETE) possibly use a suborder of the maximal order. This makes rnfpolredabs behave as rnfpolredbest. Just use the latter.

Warning. The complexity of rnfpolredabs is exponential in the absolute degree. The function rnfpolredbest runs in polynomial time, and tends to return polynomials with smaller discriminants. It also supports polynomials with arbitrary coefficients in *nf*, neither integral nor necessarily monic.

rnfpolredbest(pol, flag)

Relative version of polredbest. Given a polynomial *pol* with coefficients in nf, finds a simpler relative polynomial P defining the same field. As opposed to rnfpolredabs this function does not return a *smallest* (canonical) polynomial with respect to some measure, but it does run in polynomial time.

The binary digits of flag correspond to 1: add information to convert elements to the new representation, 2: absolute polynomial, instead of relative. More precisely:

0: default, return P

1: returns [P, a] where P is the default output and a, a t_POLMOD modulo P, is a root of pol.

2: returns Pabs, an absolute, instead of a relative, polynomial. Same as but faster than

rnfequation(nf, rnfpolredbest(nf,pol))

3: returns [Pabs, a, b], where *Pabs* is an absolute polynomial as above, a, b are t_POLMOD modulo *Pabs*, roots of nf.pol and *pol* respectively.

```
? K = nfinit(y^3-2); pol = x^2 + x^*y + y^2;
? [P, a] = rnfpolredbest(K,pol,1);
? P
\%3 = x^2 - x + Mod(y - 1, y^3 - 2)
? a
\%4 = Mod(Mod(2*y^{2}+3*y+4,y^{3}-2)*x + Mod(-y^{2}-2*y-2,y^{3}-2),
x^2 - x + Mod(y-1, y^3-2))
? subst(K.pol,y,a)
%5 = 0
? [Pabs, a, b] = rnfpolredbest(K,pol,3);
? Pabs
\%7 = x^{6} - 3^{*}x^{5} + 5^{*}x^{3} - 3^{*}x + 1
? a
\%8 = Mod(-x^2+x+1, x^6-3x^5+5x^3-3x+1)
?b
\%9 = Mod(2*x^{5}-5*x^{4}-3*x^{3}+10*x^{2}+5*x-5, x^{6}-3*x^{5}+5*x^{3}-3*x+1)
? subst(K.pol,y,a)
\%10 = 0
? substvec(pol,[x,y],[a,b])
%11 = 0
```

rnfpseudobasis(T)

Given an *nf* structure attached to a number field K, as output by nfinit, and a monic irreducible polynomial Tin $\mathbb{Z}_K[x]$ defining a relative extension L = K[x]/(T), computes the relative discriminant of L and a pseudo-basis (A, J) for the maximal order \mathbb{Z}_L viewed as a \mathbb{Z}_K -module. This is output as a vector [A, J, D, d], where D is the relative ideal discriminant and d is the relative discriminant considered as an element of K^*/K^{*2} .

```
? K = nfinit(y^2+1);
? [A,J,D,d] = rnfpseudobasis(K, x^2+y);
? A
%3 =
[1 0]
[0 1]
? J
%4 = [1, 1]
? D
%5 = [0, -4]~
? d
%6 = [0, -1]~
```

Huge discriminants, helping rnfdisc. The format [T, B] is also accepted instead of T and produce an order which is maximal at all prime ideals specified by B, see ??rnfinit.

```
? p = 585403248812100232206609398101;
? q = 711171340236468512951957953369;
? T = x^2 + 3*(p*q)^2;
? [A,J,D,d] = V = rnfpseudobasis(K, T); D
time = 22,178 ms.
%10 = 3
? [A,J,D,d] = W = rnfpseudobasis(K, [T,100]); D
time = 5 ms.
%11 = 3
? V == W
%12 = 1
? [A,J,D,d] = W = rnfpseudobasis(K, [T, [3]]); D
%13 = 3
? V == W
%14 = 1
```

In this example, the results are identical since $D \cap \mathbb{Z}$ factors over primes less than 100 (and in fact, over 3). Had it not been the case, the order would have been guaranteed maximal at primes p || p for $p \leq 100$ only (resp. p || 3). And might have been nonmaximal at any other prime ideal p such that p^2 divided D.

rnfsteinitz(x)

Given a number field nf as output by nfinit and either a polynomial x with coefficients in nf defining a relative extension L of nf, or a pseudo-basis x of such an extension as output for example by rnfpseudobasis, computes another pseudo-basis (A, I) (not in HNF in general) such that all the ideals of I except perhaps the last one are equal to the ring of integers of nf, and outputs the four-component row vector [A, I, D, d] as in rnfpseudobasis. The name of this function comes from the fact that the ideal class of the last ideal of I, which is well defined, is the Steinitz class of the \mathbb{Z}_K -module \mathbb{Z}_L (its image in $SK_0(\mathbb{Z}_K)$).

round(e)

If x is in \mathbb{R} , rounds x to the nearest integer (rounding to +oo in case of ties), then and sets e to the number of error bits, that is the binary exponent of the difference between the original and the rounded value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. e > 0), the result is undefined and an error occurs if e was not given.

Important remark. Contrary to the other truncation functions, this function operates on every coefficient at every level of a PARI object. For example

$$truncate((2.4 * X^2 - 1.7)/(X)) = 2.4 * X,$$

whereas

$$round((2.4 * X^2 - 1.7)/(X)) = (2 * X^2 - 2)/(X).$$

An important use of **round** is to get exact results after an approximate computation, when theory tells you that the coefficients must be integers.

select(A, flag)

We first describe the default behavior, when flag is 0 or omitted. Given a vector or list A and a t_CLOSURE f, select returns the elements x of A such that f(x) is nonzero. In other words, f is seen as a selection function returning a boolean value.

```
? select(x->isprime(x), vector(50,i,i^2+1))
%1 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]
? select(x->(x<100), %)
%2 = [2, 5, 17, 37]</pre>
```

returns the primes of the form $i^2 + 1$ for some $i \le 50$, then the elements less than 100 in the preceding result. The select function also applies to a matrix A, seen as a vector of columns, i.e. it selects columns instead of entries, and returns the matrix whose columns are the selected ones.

Remark. For v a t_VEC, t_COL, t_VECSMALL, t_LIST or t_MAT, the alternative set-notations

[g(x) | x <- v, f(x)] [x | x <- v, f(x)] [g(x) | x <- v]

are available as shortcuts for

```
apply(g, select(f, Vec(v)))
select(f, Vec(v))
apply(g, Vec(v))
```

respectively:

? [x | x <- vector(50,i,i^2+1), isprime(x)]
%1 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]</pre>

If flag = 1, this function returns instead the *indices* of the selected elements, and not the elements themselves (indirect selection):

```
? V = vector(50,i,i^2+1);
? select(x->isprime(x), V, 1)
%2 = Vecsmall([1, 2, 4, 6, 10, 14, 16, 20, 24, 26, 36, 40])
? vecextract(V, %)
%3 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]
```

The following function lists the elements in $(\mathbb{Z}/N\mathbb{Z})^*$:

```
? invertibles(N) = select(x->gcd(x,N) == 1, [1..N])
```

Finally

? select(x->x, M)

selects the nonzero entries in M. If the latter is a t_MAT, we extract the matrix of nonzero columns. Note that *removing* entries instead of selecting them just involves replacing the selection function f with its negation:

? select(x->!isprime(x), vector(50,i,i^2+1))

seralgdep(p, r)

finds a linear relation between powers $(1, s, ..., s^p)$ of the series s, with polynomial coefficients of degree $\langle = r$. In case no relation is found, return 0.

```
? s = 1 + 10*y - 46*y^2 + 460*y^3 - 5658*y^4 + 77740*y^5 + O(y^6);
? seralgdep(s, 2, 2)
```

The series main variable must not be x, so as to be able to express the result as a polynomial in x.

serchop(n)

Remove all terms of degree strictly less than n in series s. When the series contains no terms of degree < n, return $O(x^n)$.

```
? s = 1/x + x + 2*x<sup>2</sup> + 0(x<sup>3</sup>);
? serchop(s)
%2 = x + 2*x<sup>3</sup> + 0(x<sup>3</sup>)
? serchop(s, 2)
%3 = 2*x<sup>2</sup> + 0(x<sup>3</sup>)
? serchop(s, 100)
%4 = 0(x<sup>100</sup>)
```

serconvol(y)

Convolution (or Hadamard product) of the two power series x and y; in other words if $x = \sum a_k * X^k$ and $y = \sum b_k * X^k$ then $serconvol(x, y) = \sum a_k * b_k * X^k$.

serlaplace()

x must be a power series with nonnegative exponents or a polynomial. If $x = \sum (a_k/k!) * X^k$ then the result is $\sum a_k * X^k$.

serprec(v)

Returns the absolute precision of x with respect to power series in the variable v; this is the minimum precision of the components of x. The result is +oo if x is an exact object (as a series in v):

? serprec(x + 0(y^2), y)
%1 = 2
? serprec(x + 2, x)
%2 = +oo
? serprec(2 + x + 0(x^2), y)
%3 = +oo

serreverse()

Reverse power series of s, i.e. the series t such that t(s) = x; s must be a power series whose valuation is exactly equal to one.

? \ps 8 ? t = serreverse(tan(x)) %2 = x - $1/3*x^3 + 1/5*x^5 - 1/7*x^7 + 0(x^8)$? tan(t) %3 = x + $0(x^8)$

setbinop(X, Y)

The set whose elements are the f(x,y), where x,y run through X,Y. respectively. If Y is omitted, assume that X = Y and that f is symmetric: f(x,y) = f(y,x) for all x, y in X.

? X = [1,2,3]; Y = [2,3,4]; ? setbinop((x,y)->x+y, X,Y) \\ set X + Y %2 = [3, 4, 5, 6, 7] ? setbinop((x,y)->x-y, X,Y) \\ set X - Y %3 = [-3, -2, -1, 0, 1] ? setbinop((x,y)->x+y, X) \\ set 2X = X + X %2 = [2, 3, 4, 5, 6]

setintersect(y)

Intersection of the two sets x and y (see setisset). If x or y is not a set, the result is undefined.

setisset()

Returns true (1) if x is a set, false (0) if not. In PARI, a set is a row vector whose entries are strictly increasing with respect to a (somewhat arbitrary) universal comparison function. To convert any object into a set (this is most useful for vectors, of course), use the function Set.

```
? a = [3, 1, 1, 2];
? setisset(a)
%2 = 0
? Set(a)
%3 = [1, 2, 3]
```

setminus(y)

Difference of the two sets x and y (see setisset), i.e. set of elements of x which do not belong to y. If x or y is not a set, the result is undefined.

setrand()

Reseeds the random number generator using the seed n. No value is returned. The seed is a small positive integer $0 < n < 2^{64}$ used to generate deterministically a suitable state array. All gp session start by an implicit setrand(1), so resetting the seed to this value allows to replay all computations since the session start. Alternatively, running a randomized computation starting by setrand (n) twice with the same n will generate the exact same output.

In the other direction, including a call to setrand(getwalltime()) from your gprc will cause GP to produce different streams of random numbers in each session. (Unix users may want to use /dev/urandom instead of getwalltime.)

For debugging purposes, one can also record a particular random state using getrand (the value is encoded as a huge integer) and feed it to setrand:

```
? state = getrand(); \\ record seed
...
? setrand(state); \\ we can now replay the exact same computations
```

setsearch(x, flag)

Determines whether x belongs to the set S (see setisset).

We first describe the default behavior, when flag is zero or omitted. If x belongs to the set S, returns the index j such that S[j] = x, otherwise returns 0.

? T = [7,2,3,5]; S = Set(T); ? setsearch(S, 2) %2 = 1 ? setsearch(S, 4) \\ not found %3 = 0

```
? setsearch(T, 7) \setminus search in a randomly sorted vector %4 = 0 \setminus WRONG !
```

If S is not a set, we also allow sorted lists with respect to the cmp sorting function, without repeated entries, as per listsort(L, 1); otherwise the result is undefined.

```
? L = List([1,4,2,3,2]); setsearch(L, 4)
%1 = 0 \\ WRONG !
? listsort(L, 1); L \\ sort L first
%2 = List([1, 2, 3, 4])
? setsearch(L, 4)
%3 = 4 \\ now correct
```

If flag is nonzero, this function returns the index j where x should be inserted, and 0 if it already belongs to S. This is meant to be used for dynamically growing (sorted) lists, in conjunction with listinsert.

```
? L = List([1,5,2,3,2]); listsort(L,1); L
%1 = List([1,2,3,5])
? j = setsearch(L, 4, 1) \\ 4 should have been inserted at index j
%2 = 4
? listinsert(L, 4, j); L
%3 = List([1, 2, 3, 4, 5])
```

setunion(y)

Union of the two sets x and y (see setisset). If x or y is not a set, the result is undefined.

shift(n)

Shifts x componentwise left by n bits if $n \ge 0$ and right by ||n|| bits if n < 0. May be abbreviated as x :literal: <<`n or x :literal:`>>`(-n). A left shift by n corresponds to multiplication by 2^n . A right shift of an integer x by ||n|| corresponds to a Euclidean division of x by $2^{||n||}$ with a remainder of the same sign as x, hence is not the same (in general) as $x 2^n$.

shiftmul(n)

Multiplies x by 2^n . The difference with shift is that when n < 0, ordinary division takes place, hence for example if x is an integer the result may be a fraction, while for shifts Euclidean division takes place when n < 0 hence if x is an integer the result is still an integer.

sigma(k)

Sum of the k - th powers of the positive divisors of ||x||. x and k must be of type integer.

sign()

sign (0, 1 or -1) of x, which must be of type integer, real or fraction; t_QUAD with positive discriminants and t_INFINITY are also supported.

simplify()

This function simplifies x as much as it can. Specifically, a complex or quadratic number whose imaginary part is the integer 0 (i.e. not Mod(0, 2) or 0.E-28) is converted to its real part, and a polynomial of degree 0 is converted to its constant term. Simplifications occur recursively.

This function is especially useful before using arithmetic functions, which expect integer arguments:

? x = 2 + y - y %1 = 2 ? isprime(x)

```
*** at top-level: isprime(x)
*** ^-----
*** isprime: not an integer argument in an arithmetic function
? type(x)
%2 = "t_POL"
? type(simplify(x))
%3 = "t_INT"
```

Note that GP results are simplified as above before they are stored in the history. (Unless you disable automatic simplification with \backslash y, that is.) In particular

? type(%1)
%4 = "t_INT"

sin(precision)

Sine of x. Note that, for real x, cosine and sine can be obtained simultaneously as

```
cs(x) = my(z = exp(I*x)); [real(z), imag(z)];
```

and for general complex x as

 $cs2(x) = my(z = exp(I^*x), u = 1/z); [(z+u)/2, (z-u)/2];$

Note that the latter function suffers from catastrophic cancellation when z^2 1.

sinc(precision)

Cardinal sine of x, i.e. $\sin(x)/x$ if x! = 0, 1 otherwise. Note that this function also allows to compute

$$(1 - \cos(x))/x^2 = sinc(x/2)^2/2$$

accurately near x = 0.

sinh(precision)

Hyperbolic sine of x.

sizebyte()

Outputs the total number of bytes occupied by the tree representing the PARI object x.

sizedigit()

This function is DEPRECATED, essentially meaningless, and provided for backwards compatibility only. Don't use it!

outputs a quick upper bound for the number of decimal digits of (the components of) x, off by at most 1. More precisely, for a positive integer x, it computes (approximately) the ceiling of

$$floor(1 + \log_2 x) \log_{10} 2,$$

To count the number of decimal digits of a positive integer x, use #digits(x). To estimate (recursively) the size of x, use normlp(x).

sqr()

Square of x. This operation is not completely straightforward, i.e. identical to x * x, since it can usually be computed more efficiently (roughly one-half of the elementary multiplications can be saved). Also, squaring a 2-adic number increases its precision. For example,

? $(1 + 0(2^4))^2$ %1 = 1 + 0(2^5) ? $(1 + 0(2^4)) * (1 + 0(2^4))$ %2 = 1 + 0(2^4)

Note that this function is also called whenever one multiplies two objects which are known to be *identical*, e.g. they are the value of the same variable, or we are computing a power.

? x = (1 + 0(2⁴)); x * x %3 = 1 + 0(2⁵) ? (1 + 0(2⁴))⁴ %4 = 1 + 0(2⁶)

(note the difference between %2 and %3 above).

sqrt(precision)

Principal branch of the square root of x, defined as $\sqrt{x} = \exp(\log x/2)$. In particular, we have $Arg(sqrt(x)) \in [-\pi/2, \pi/2]$, and if $x \in \mathbb{R}$ and x < 0, then the result is complex with positive imaginary part.

Intmod a prime p, t_PADIC and t_FFELT are allowed as arguments. In the first 2 cases (t_INTMOD, t_PADIC), the square root (if it exists) which is returned is the one whose first p-adic digit is in the interval [0, p/2]. For other arguments, the result is undefined.

sqrtint(r)

Returns the integer square root of x, i.e. the largest integer y such that $y^2 \le x$, where x a nonnegative integer. If r is present, set it to the remainder $r = x - y^2$, which satisfies $0 \le r \le 2y$

```
? x = 120938191237; sqrtint(x)
%1 = 347761
? sqrt(x)
%2 = 347761.68741970412747602130964414095216
? y = sqrtint(x, &r)
%3 = 347761
? x - y^2
%4 = 478116
```

sqrtn(n, z, precision)

Principal branch of the *n*, i.e. such that $Arg(sqrtn(x)) \in]-\pi/n, \pi/n]$. Intmod a prime and *p*-adics are allowed as arguments.

If z is present, it is set to a suitable root of unity allowing to recover all the other roots. If it was not possible, z is set to zero. In the case this argument is present and no n is returned instead of raising an error.

```
? sqrtn(Mod(2,7), 2)
%1 = Mod(3, 7)
? sqrtn(Mod(2,7), 2, &z); z
%2 = Mod(6, 7)
? sqrtn(Mod(2,7), 3)
*** at top-level: sqrtn(Mod(2,7),3)
*** ^------
*** sqrtn: nth-root does not exist in gsqrtn.
? sqrtn(Mod(2,7), 3, &z)
%2 = 0
? z
%3 = 0
```

The following script computes all roots in all possible cases:

```
sqrtnall(x,n)=
{ my(V,r,z,r2);
    r = sqrtn(x,n, &z);
    if (!z, error("Impossible case in sqrtn"));
    if (type(x) == "t_INTMOD" || type(x)=="t_PADIC",
    r2 = r*z; n = 1;
    while (r2!=r, r2*=z;n++));
    V = vector(n); V[1] = r;
    for(i=2, n, V[i] = V[i-1]*z);
    V
}
addhelp(sqrtnall,"sqrtnall(x,n):compute the vector of nth-roots of x");
```

sqrtnint(n)

Returns the integer n-th root of x, i.e. the largest integer y such that $y^n \le x$, where x is a nonnegative integer.

```
? N = 120938191237; sqrtnint(N, 5)
%1 = 164
? N^(1/5)
%2 = 164.63140849829660842958614676939677391
```

The special case n = 2 is sqrtint

strchr()

Converts integer or vector of integers x to a string, translating each integer (in the range [1, 255]) into a character using ASCII encoding.

```
? strchr(97)
%1 = "a"
? Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100])
? strchr(%)
%3 = "hello world"
```

strjoin(p)

Joins the strings in vector v, separating them with delimiter p. The reverse operation is strsplit.

```
? v = ["abc", "def", "ghi"]
? strjoin(v, "/")
%2 = "abc/def/ghi"
? strjoin(v)
%3 = "abcdefghi"
```

strsplit(p)

Splits the string s into a vector of strings, with p acting as a delimiter. If p is empty or omitted, split the string into characters.

```
? strsplit("abc::def::ghi", "::")
%1 = ["abc", "def", "ghi"]
? strsplit("abc", "")
%2 = ["a", "b", "c"]
? strsplit("aba", "a")
```

If s starts (resp. ends) with the pattern p, then the first (resp. last) entry in the vector is the empty string:

? strsplit("aba", "a")
%3 = ["", "b", ""]

subgrouplist(bound, flag)

cyc being a vector of positive integers giving the cyclic components for a finite Abelian group G (or any object which has a .cyc method), outputs the list of subgroups of G. Subgroups are given as HNF left divisors of the SNF matrix corresponding to G.

If flag = 0 (default) and *cyc* is a *bnr* structure output by bnrinit, gives only the subgroups whose modulus is the conductor. Otherwise, all subgroups are given.

If *bound* is present, and is a positive integer, restrict the output to subgroups of index less than *bound*. If *bound* is a vector containing a single positive integer B, then only subgroups of index exactly equal to B are computed. For instance

```
? subgrouplist([6,2])
%1 = [[6, 0; 0, 2], [2, 0; 0, 2], [6, 3; 0, 1], [2, 1; 0, 1], [3, 0; 0, 2],
[1, 0; 0, 2], [6, 0; 0, 1], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],3) \\ index less than 3
%2 = [[2, 1; 0, 1], [1, 0; 0, 2], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],[3]) \\ index 3
%3 = [[3, 0; 0, 1]]
? bnr = bnrinit(bnfinit(x), [120,[1]], 1);
? L = subgrouplist(bnr, [8]);
```

In the last example, L corresponds to the 24 subfields of $\mathbb{Q}(\zeta_{120})$, of degree 8 and conductor 12000 (by setting *flag*, we see there are a total of 43 subgroups of degree 8).

```
? vector(#L, i, galoissubcyclo(bnr, L[i]))
```

will produce their equations. (For a general base field, you would have to rely on bnrstark, or bnrclassfield.)

Warning. This function requires factoring the exponent of G. If you are only interested in subgroups of index n (or dividing n), you may considerably speed up the function by computing the subgroups of G/G^n , whose cyclic components are apply(x - > gcd(n,x), C) (where C gives the cyclic components of G). If you want the *bnr* variant, now is a good time to use bnrinit(,,, n) as well, to directly compute the ray class group modulo n-th powers.

subst(y, z)

Replace the simple variable y by the argument z in the "polynomial" expression x. If z is a vector, return the vector of the evaluated expressions subst(x, y, z[i]).

Every type is allowed for x, but if it is not a genuine polynomial (or power series, or rational function), the substitution will be done as if the scalar components were polynomials of degree zero. In particular, beware that:

```
? subst(1, x, [1,2; 3,4])
%1 =
[1 0]
[0 1]
? subst(1, x, Mat([0,1]))
*** at top-level: subst(1,x,Mat([0,1])
*** ^------
*** subst: forbidden substitution by a non square matrix.
```

If x is a power series, z must be either a polynomial, a power series, or a rational function. If x is a vector, matrix or list, the substitution is applied to each individual entry.

Use the function substvec to replace several variables at once, or the function substpol to replace a polynomial expression.

substpol(y, z)

Replace the "variable" y by the argument z in the "polynomial" expression x. Every type is allowed for x, but the same behavior as subst above apply.

The difference with subst is that y is allowed to be any polynomial here. The substitution is done moding out all components of x (recursively) by y - t, where t is a new free variable of lowest priority. Then substituting t by z in the resulting expression. For instance

```
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, x<sup>2</sup>, y)
%1 = y<sup>2</sup> + y + 1
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, x<sup>3</sup>, y)
%2 = x<sup>2</sup> + y<sup>*</sup>x + 1
? substpol(x<sup>4</sup> + x<sup>2</sup> + 1, (x+1)<sup>2</sup>, y)
%3 = (-4<sup>*</sup>y - 6)<sup>*</sup>x + (y<sup>2</sup> + 3<sup>*</sup>y - 3)
```

substvec(v, w)

v being a vector of monomials of degree 1 (variables), w a vector of expressions of the same length, replace in the expression x all occurrences of v_i by w_i . The substitutions are done simultaneously; more precisely, the v_i are first replaced by new variables in x, then these are replaced by the w_i :

```
? substvec([x,y], [x,y], [y,x])
%1 = [y, x]
? substvec([x,y], [x,y], [y,x+y])
%2 = [y, x + y] \\ not [y, 2*y]
```

sumdedekind(k)

Returns the Dedekind sum attached to the integers h and k, corresponding to a fast implementation of

s(h,k) = sum(n = 1, k-1, (n/k)*(frac(h*n/k) - 1/2))

sumdigits(B)

Sum of digits in the integer ||n||, when written in base B > 1.

```
? sumdigits(123456789)
%1 = 45
? sumdigits(123456789, 2)
%1 = 16
```

Note that the sum of bits in n is also returned by hammingweight. This function is much faster than vecsum(digits(n,B)) when B is 10 or a power of 2, and only slightly faster in other cases.

```
sumeulerrat(s, a, precision)
```

 $\sum_{p>=a} F(p^s)$, where the sum is taken over prime numbers and F is a rational function.

```
? sumeulerrat(1/p^2)
%1 = 0.45224742004106549850654336483224793417
? sumeulerrat(1/p, 2)
%2 = 0.45224742004106549850654336483224793417
```

sumformal(v)

formal sum of the polynomial expression f with respect to the main variable if v is omitted, with respect to the variable v otherwise; it is assumed that the base ring has characteristic zero. In other words, considering f as a polynomial function in the variable v, returns F, a polynomial in v vanishing at 0, such that $F(b) - F(a) = sum_{v=a+1}^{b} f(v)$:

```
? sumformal(n) \setminus 1 + ... + n

\%1 = 1/2*n^2 + 1/2*n

? f(n) = n^3+n^2+1;

? F = sumformal(f(n)) \setminus f(1) + ... + f(n)

\%3 = 1/4*n^4 + 5/6*n^3 + 3/4*n^2 + 7/6*n

? sum(n = 1, 2000, f(n)) == subst(F, n, 2000)

\%4 = 1

? sum(n = 1001, 2000, f(n)) == subst(F, n, 2000) - subst(F, n, 1000)

\%5 = 1

? sumformal(x^2 + x*y + y^2, y)

\%6 = y*x^2 + (1/2*y^2 + 1/2*y)*x + (1/3*y^3 + 1/2*y^2 + 1/6*y)

? x^2 * y + x * sumformal(y) + sumformal(y^2) == %

\%7 = 1
```

sumnumapinit(precision)

Initialize tables for Abel-Plana summation of a series $\sum f(n)$, where f is holomorphic in a right half-plane. If given, asymp is of the form $[+oo, \alpha]$, as in intrum and indicates the decrease rate at infinity of functions to be summed. A positive $\alpha > 0$ encodes an exponential decrease of type $\exp(-\alpha n)$ and a negative $-2 < \alpha < -1$ encodes a slow polynomial decrease of type n^{α} .

```
? \p200
? sumnumap(n=1, n^{-2});
time = 163 \text{ ms}.
? tab = sumnumapinit();
time = 160 \text{ ms.}
? sumnumap(n=1, n^-2, tab); \\ faster
time = 7 \text{ ms}.
? tab = sumnumapinit([+oo, log(2)]); \\ decrease like 2^-n
time = 164 \text{ ms.}
? sumnumap(n=1, 2^-n, tab) - 1
time = 36 \text{ ms}.
%5 = 3.0127431466707723218 E-282
? tab = sumnumapinit([+oo, -4/3]); \\ decrease like n^(-4/3)
time = 166 \text{ ms.}
? sumnumap(n=1, n^(-4/3), tab);
time = 181 \text{ ms.}
```

sumnuminit(precision)

Initialize tables for Euler-MacLaurin delta summation of a series with positive terms. If given, asymp is of the form $[+oo, \alpha]$, as in intrum and indicates the decrease rate at infinity of functions to be summed. A positive $\alpha > 0$ encodes an exponential decrease of type $\exp(-\alpha n)$ and a negative $-2 < \alpha < -1$ encodes a slow polynomial decrease of type n^{α} .

? \p200 ? sumnum(n=1, n^-2);

```
time = 200 ms.
? tab = sumnuminit();
time = 188 ms.
? sumnum(n=1, n^-2, tab); \\ faster
time = 8 ms.
? tab = sumnuminit([+oo, log(2)]); \\ decrease like 2^-n
time = 200 ms.
? sumnum(n=1, 2^-n, tab)
time = 44 ms.
? tab = sumnuminit([+oo, -4/3]); \\ decrease like n^(-4/3)
time = 200 ms.
? sumnum(n=1, n^(-4/3), tab);
time = 221 ms.
```

sumnumlagrangeinit(c1, precision)

Initialize tables for Lagrange summation of a series. By default, assume that the remainder $R(n) = \sum_{m>=n} f(m)$ has an asymptotic expansion

$$R(n) = \sum_{m>=n} f(n) \sum_{i>=1} a_i / n^i$$

at infinity. The argument asymp allows to specify different expansions:

• a real number β means

$$R(n) = n^{-\beta} \sum_{i>=1} a_i / n^i$$

• a t_CLOSURE g means

$$R(n) = g(n) \sum_{i>=1} a_i / n^i$$

 $(The preceding case corresponds to: math: `g(n) = n^{-\beta}`.)$

• a pair $[\alpha, \beta]$ where β is as above and $\alpha \in 2, 1, 1/2, 1/3, 1/4$. We let $R_2(n) = R(n) - f(n)/2$ and $R_\alpha(n) = R(n)$ for $\alpha! = 2$. Then

 $R_{\alpha}(n) = g(n) \sum_{i>=1} a_i / n^{i\alpha}$

 $Note that the initialization time sincrease considerable for the: math: `\alpha` is this list(: math: `1/4` being the slowest).$

The constant c1 is technical and computed by the program, but can be set by the user: the number of interpolation steps will be chosen close to c1.B, where B is the bit accuracy.

```
? \p2000
? sumnumlagrange(n=1, n^-2);
time = 173 ms.
? tab = sumnumlagrangeinit();
time = 172 ms.
? sumnumlagrange(n=1, n^-2, tab);
time = 4 ms.
```

```
? \p115
? sumnumlagrange(n=1, n^{-4/3}) - zeta(4/3);
%1 = -0.1093[...] \setminus junk: expansion in n^{(1/3)}
time = 84 \text{ ms}.
? tab = sumnumlagrangeinit([1/3,0]); \\ alpha = 1/3
time = 336 \text{ ms.}
? sumnumlagrange(n=1, n^(-4/3), tab) - zeta(4/3)
time = 84 \text{ ms}.
%3 = 1.0151767349262596893 E-115 \\ now OK
? tab = sumnumlagrangeinit(1/3); \\ alpha = 1, beta = 1/3: much faster
time = 3ms
? sumnumlagrange(n=1, n^(-4/3), tab) - zeta(4/3) \setminus ... but wrong
\%5 = -0.273825[...] \setminus junk !
? tab = sumnumlagrangeinit(-2/3); \\ alpha = 1, beta = -2/3
time = 3ms
? sumnumlagrange(n=1, n^{-4/3}, tab) - zeta(4/3)
%6 = 2.030353469852519379 E-115 \\ now OK
```

in The final example with $\zeta(4/3)$, the remainder $R_1(n)$ is of the form $n^{-1/3} \sum_{i>=0} a_i/n^i$, i.e. $n^{2/3} \sum_{i>=1} a_i/n^i$. The explains the wrong result for $\beta = 1/3$ and the correction with $\beta = -2/3$.

sumnummonieninit(w, n0, precision)

Initialize tables for Monien summation of a series $\sum_{n>=n_0} f(n)$ where f(1/z) has a complex analytic continuation in a (complex) neighbourhood of the segment [0, 1].

By default, assume that $f(n) = O(n^{-2})$ and has a nonzero asymptotic expansion

$$f(n) = \sum_{i>=2} a_i/n^2$$

at infinity. Note that the sum starts at i = 2! The argument asymp allows to specify different expansions:

• a real number $\beta > 0$ means

$$f(n) = \sum_{i > = 1} a_i / n^{i + \beta}$$
 (Now
the summation starts at : math : '1'.)

• a vector $[\alpha, \beta]$ of reals, where we must have $\alpha > 0$ and $\alpha + \beta > 1$ to ensure convergence, means that

$$f(n) = \sum_{i>=1} a_i / n^{\alpha i + \beta}$$

 $Note that: math: `asymp = [1, \beta]` is equivalent to: math: `asymp = \beta`.$

```
? \p57
? s = sumnum(n = 1, sin(1/sqrt(n)) / n); \\ reference point
? \p38
? sumnummonien(n = 1, sin(1/sqrt(n)) / n) - s
%2 = -0.001[...] \\ completely wrong
```

```
? t = sumnummonieninit(1/2); \\ f(n) = sum_i 1 / n^(i+1/2)
? sumnummonien(n = 1, sin(1/sqrt(n)) / n, t) - s
%3 = 0.E-37 \\ now correct
```

(As a matter of fact, in the above summation, the result given by summum at p38 is slighly incorrect, so we had to increase the accuracy to p57.)

The argument w is used to sum expressions of the form

$$\sum_{n>=n_0} f(n)w(n),$$

for varying f as above, and fixed weight function w, where we further assume that the auxiliary sums

$$g_w(m) = \sum_{n > = n_0} w(n) / n^{\alpha m + \beta}$$

converge for all $m \ge 1$. Note that for nonnegative integers k, and weight $w(n) = (\log n)^k$, the function $g_w(m) = \zeta^{(k)}(\alpha m + \beta)$ has a simple expression; for general weights, g_w is computed using summum. The following variants are available

- an integer $k \ge 0$, to code $w(n) = (\log n)^k$;
- a t_CLOSURE computing the values w(n), where we assume that $w(n) = O(n^{\epsilon})$ for all $\epsilon > 0$;
- a vector [w, fast], where w is a closure as above and fast is a scalar; we assume that $w(n) = O(n^{fast+\epsilon})$; note that w = [w, 0] is equivalent to w = w. Note that if w decreases exponentially, suminf should be used instead.

The subsequent calls to sumnummonien *must* use the same value of n_0 as was used here.

```
? \p300
? sumnummonien(n = 1, n^-2*log(n)) + zeta'(2)
time = 328 ms.
%1 = -1.323[...]E-6 \\ completely wrong, f does not satisfy hypotheses !
? tab = sumnummonieninit(, 1); \\ codes w(n) = log(n)
time = 3,993 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 41 ms.
%3 = -5.562684646268003458 E-309 \\ now perfect
? tab = sumnummonieninit(, n->log(n)); \\ generic, slower
time = 9,808 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 40 ms.
%5 = -5.562684646268003458 E-309 \\ identical result
```

sumnumrat(a, precision)

 $\sum_{n>=a} F(n)$, where F is a rational function of degree less than or equal to -2 and where poles of F at integers >=a are omitted from the summation. The argument a must be a t_INT or -oo.

? sumnumrat(1/(x²+1)²,0) %1 = 1.3068369754229086939178621382829073480 ? sumnumrat(1/x², -oo) \\ value at x=0 is discarded %2 = 3.2898681336964528729448303332920503784

? 2*zeta(2)
%3 = 3.2898681336964528729448303332920503784

When deg F = -1, we define

$$\sum_{-oo}^{oo} F(n) := \sum_{n \ge 0} (F(n) + F(-1 - n)) :$$

? sumnumrat(1/x, -oo) %4 = 0.E-38

tan(precision)

Tangent of x.

tanh(precision)

Hyperbolic tangent of x.

taylor(t, serprec)

Taylor expansion around 0 of x with respect to the simple variable t. x can be of any reasonable type, for example a rational function. Contrary to Ser, which takes the valuation into account, this function adds $O(t^d)$ to all components of x.

```
? taylor(x/(1+y), y, 5)
%1 = (y^4 - y^3 + y^2 - y + 1)*x + 0(y^5)
? Ser(x/(1+y), y, 5)
*** at top-level: Ser(x/(1+y),y,5)
*** ^------
*** Ser: main variable must have higher priority in gtoser.
```

teichmuller(tab)

Teichmüller character of the *p*-adic number *x*, i.e. the unique (p-1)-th root of unity congruent to $x/p^{v_p(x)}$ modulo *p*. If *x* is of the form [p, n], for a prime *p* and integer *n*, return the lifts to \mathbb{Z} of the images of $i + O(p^n)$ for i = 1, ..., p - 1, i.e. all roots of 1 ordered by residue class modulo *p*. Such a vector can be fed back to teichmuller, as the optional argument tab, to speed up later computations.

```
? z = teichmuller(2 + 0(101^5))
%1 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
? z^100
%2 = 1 + 0(101^5)
? T = teichmuller([101, 5]);
? teichmuller(2 + 0(101^5), T)
%4 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
```

As a rule of thumb, if more than

 $p/2(\log_2(p) + hammingweight(p))$

values of teichmuller are to be computed, then it is worthwile to initialize:

```
? p = 101; n = 100; T = teichmuller([p,n]); \\ instantaneous
? for(i=1,10^3, vector(p-1, i, teichmuller(i+0(p^n), T)))
time = 60 ms.
```

```
? for(i=1,10^3, vector(p-1, i, teichmuller(i+0(p^n))))
time = 1,293 ms.
? 1 + 2*(log(p)/log(2) + hammingweight(p))
%8 = 22.316[...]
```

Here the precomputation induces a speedup by a factor 1293/60 21.5.

Caveat. If the accuracy of tab (the argument n above) is lower than the precision of x, the *former* is used, i.e. the cached value is not refined to higher accuracy. It the accuracy of tab is larger, then the precision of x is used:

```
? Tlow = teichmuller([101, 2]); \\ lower accuracy !
? teichmuller(2 + 0(101^5), Tlow)
%10 = 2 + 83*101 + 0(101^5) \\ no longer a root of 1
? Thigh = teichmuller([101, 10]); \\ higher accuracy
? teichmuller(2 + 0(101^5), Thigh)
%12 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + 0(101^5)
```

theta(*z*, *precision*)

Jacobi sine theta-function

$$\theta_1(z,q) = 2q^{1/4} \sum_{n>=0} (-1)^n q^{n(n+1)} \sin((2n+1)z).$$

thetanullk(k, precision)

k-th derivative at z = 0 of theta(q, z).

thue(a, sol)

Returns all solutions of the equation P(x, y) = a in integers x and y, where *tnf* was created with *thueinit*(P). If present, *sol* must contain the solutions of Norm(x) = a modulo units of positive norm in the number field defined by P (as computed by bnfisintnorm). If there are infinitely many solutions, an error is issued.

It is allowed to input directly the polynomial P instead of a *tnf*, in which case, the function first performs thueinit(P, \emptyset). This is very wasteful if more than one value of a is required.

If *tnf* was computed without assuming GRH (flag 1 in thueinit), then the result is unconditional. Otherwise, it depends in principle of the truth of the GRH, but may still be unconditionally correct in some favorable cases. The result is conditional on the GRH if a! = 1 and P has a single irreducible rational factor, whose attached tentative class number h and regulator R (as computed assuming the GRH) satisfy

• *h* > 1,

•
$$R/0.2 > 1.5$$

Here's how to solve the Thue equation $x^{13} - 5y^{13} = -4$:

? tnf = thueinit(x^13 - 5); ? thue(tnf, -4) %1 = [[1, 1]]

In this case, one checks that $bnfinit(x^{13} -5)$.no is 1. Hence, the only solution is (x, y) = (1, 1) and the result is unconditional. On the other hand:

```
? P = x^3-2*x^2+3*x-17; tnf = thueinit(P);
? thue(tnf, -15)
```

```
%2 = [[1, 1]] \\ a priori conditional on the GRH.
? K = bnfinit(P); K.no
%3 = 3
? K.reg
%4 = 2.8682185139262873674706034475498755834
```

This time the result is conditional. All results computed using this particular *tnf* are likewise conditional, *except* for a right-hand side of 1. The above result is in fact correct, so we did not just disprove the GRH:

```
? tnf = thueinit(x^3-2*x^2+3*x-17, 1 /*unconditional*/);
? thue(tnf, -15)
%4 = [[1, 1]]
```

Note that reducible or nonmonic polynomials are allowed:

```
? tnf = thueinit((2*x+1)^5 * (4*x^3-2*x^2+3*x-17), 1);
? thue(tnf, 128)
%2 = [[-1, 0], [1, 0]]
```

Reducible polynomials are in fact much easier to handle.

Note. When P is irreducible without a real root, the default strategy is to use brute force enumeration in time $||a||^{1/\deg P}$ and avoid computing a tough *bnf* attached to P, see thueinit. Besides reusing a quantity you might need for other purposes, the default argument *sol* can also be used to use a different strategy and prove that there are no solutions; of course you need to compute a *bnf* on you own to obtain *sol*. If there *are* solutions this won't help unless P is quadratic, since the enumeration will be performed in any case.

thueinit(flag, precision)

Initializes the *tnf* corresponding to P, a nonconstant univariate polynomial with integer coefficients. The result is meant to be used in conjunction with thue to solve Thue equations $P(X/Y)Y^{\deg P} = a$, where a is an integer. Accordingly, P must either have at least two distinct irreducible factors over \mathbb{Q} , or have one irreducible factor T with degree > 2 or two conjugate complex roots: under these (necessary and sufficient) conditions, the equation has finitely many integer solutions.

```
? S = thueinit(t^2+1);
? thue(S, 5)
%2 = [[-2, -1], [-2, 1], [-1, -2], [-1, 2], [1, -2], [1, 2], [2, -1], [2, 1]]
? S = thueinit(t+1);
*** at top-level: thueinit(t+1)
*** ^------
*** thueinit: domain error in thueinit: P = t + 1
```

The hardest case is when $\deg P > 2$ and P is irreducible with at least one real root. The routine then uses Bilu-Hanrot's algorithm.

If flag is nonzero, certify results unconditionally. Otherwise, assume GRH, this being much faster of course. In the latter case, the result may still be unconditionally correct, see thue. For instance in most cases where P is reducible (not a pure power of an irreducible), *or* conditional computed class groups are trivial *or* the right hand side is 1, then results are unconditional.

Note. The general philosophy is to disprove the existence of large solutions then to enumerate bounded solutions naively. The implementation will overflow when there exist huge solutions and the equation has degree > 2 (the quadratic imaginary case is special, since we can stick to **bnfisintnorm**, there are no fundamental units):

```
? thue(t^3+2, 10^30)
  *** at top-level: L=thue(t^3+2,10^30)
  *** ^------
  *** thue: overflow in thue (SmallSols): y <= 80665203789619036028928.
? thue(x^2+2, 10^30) \\ quadratic case much easier
%1 = [[-10000000000000, 0], [1000000000000, 0]]</pre>
```

Note. It is sometimes possible to circumvent the above, and in any case obtain an important speed-up, if you can write $P = Q(x^d)$ for some d > 1 and Q still satisfying the thueinit hypotheses. You can then solve the equation attached to Q then eliminate all solutions (x, y) such that either x or y is not a d-th power.

```
? thue(x^4+1, 10^40); \\ stopped after 10 hours
? filter(L,d) =
my(x,y); [[x,y] | v<-L, ispower(v[1],d,&x)&&ispower(v[2],d,&y)];
? L = thue(x^2+1, 10^40);
? filter(L, 2)
%4 = [[0, 10000000000], [1000000000, 0]]
```

The last 2 commands use less than 20ms.

Note. When P is irreducible without a real root, the equation can be solved unconditionnally in time $||a||^{1/\deg P}$. When this latter quantity is huge and the equation has no solutions, this fact may still be ascertained via arithmetic conditions but this now implies solving norm equations, computing a *bnf* and possibly assuming the GRH. When there is no real root, the code does not compute a *bnf* (with certification if flag = 1) if it expects this to be an "easy" computation (because the result would only be used for huge values of a). See thue for a way to compute an expensive *bnf* on your own and still get a result where this default cheap strategy fails.

trace()

This applies to quite general x. If x is not a matrix, it is equal to the sum of x and its conjugate, except for polmods where it is the trace as an algebraic number.

For x a square matrix, it is the ordinary trace. If x is a nonsquare matrix (but not a vector), an error occurs.

truncate(e)

Truncates x and sets e to the number of error bits. When x is in \mathbb{R} , this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and the truncated value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. e > 0), the result is undefined and an error occurs if e was not given. The function applies componentwise on vector / matrices; e is then the maximal number of error bits. If x is a rational function, the result is the "integer part" (Euclidean quotient of numerator by denominator) and e is not set.

Note a very special use of truncate: when applied to a power series, it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

type()

This is useful only under gp. Returns the internal type name of the PARI object x as a string. Check out existing type names with the metacommand t. For example type(1) will return "t_INT".

valuation(p)

Computes the highest exponent of p dividing x. If p is of type integer, x must be an integer, an intmod whose modulus is divisible by p, a fraction, a q-adic number with q = p, or a polynomial or power series in which case the valuation is the minimum of the valuation of the coefficients.

If p is of type polynomial, x must be of type polynomial or rational function, and also a power series if x is a monomial. Finally, the valuation of a vector, complex or quadratic number is the minimum of the component valuations.

If x = 0, the result is +oo if x is an exact object. If x is a p-adic numbers or power series, the result is the exponent of the zero. Any other type combinations gives an error.

variable()

Gives the main variable of the object x (the variable with the highest priority used in x), and p if x is a p-adic number. Return 0 if x has no variable attached to it.

```
? variable(x^2 + y)
%1 = x
? variable(1 + 0(5^2))
%2 = 5
? variable([x,y,z,t])
%3 = x
? variable(1)
%4 = 0
```

The construction

```
if (!variable(x),...)
```

can be used to test whether a variable is attached to x.

If x is omitted, returns the list of user variables known to the interpreter, by order of decreasing priority. (Highest priority is initially x, which come first until varhigher is used.) If varhigher or varlower are used, it is quite possible to end up with different variables (with different priorities) printed in the same way: they will then appear multiple times in the output:

```
? varhigher("y");
? varlower("y");
? variable()
%4 = [y, x, y]
```

Using v = variable() then v[1], v[2], etc. allows to recover and use existing variables.

variables()

Returns the list of all variables occurring in object x (all user variables known to the interpreter if x is omitted), sorted by decreasing priority.

? variables([x^2 + y*z + 0(t), a+x])
%1 = [x, y, z, t, a]

The construction

if (!variables(x),...)

can be used to test whether a variable is attached to x.

If varhigher or varlower are used, it is quite possible to end up with different variables (with different priorities) printed in the same way: they will then appear multiple times in the output:

```
? y1 = varhigher("y");
? y2 = varlower("y");
? variables(y*y1*y2)
%4 = [y, y, y]
```

vecextract(y, z)

Extraction of components of the vector or matrix x according to y. In case x is a matrix, its components are the *columns* of x. The parameter y is a component specifier, which is either an integer, a string describing a range, or a vector.

If y is an integer, it is considered as a mask: the binary bits of y are read from right to left, but correspond to taking the components from left to right. For example, if $y = 13 = (1101)_2$ then the components 1,3 and 4 are extracted.

If y is a vector (t_VEC, t_COL or t_VECSMALL), which must have integer entries, these entries correspond to the component numbers to be extracted, in the order specified.

If y is a string, it can be

- a single (nonzero) index giving a component number (a negative index means we start counting from the end).
- a range of the form ":math:`a..:math:b", where :math:`a and b are indexes as above. Any of a and b can be omitted; in this case, we take as default values a = 1 and b = -1, i.e. the first and last components respectively. We then extract all components in the interval [a, b], in reverse order if b < a.

In addition, if the first character in the string is ^, the complement of the given set of indices is taken.

If z is not omitted, x must be a matrix. y is then the row specifier, and z the column specifier, where the component specifier is as explained above.

```
? v = [a, b, c, d, e];
? vecextract(v, 5) \\ mask
%1 = [a, c]
? vecextract(v, [4, 2, 1]) \\ component list
%2 = [d, b, a]
? vecextract(v, "2..4") \\ interval
%3 = [b, c, d]
? vecextract(v, "-1..-3") \\ interval + reverse order
%4 = [e, d, c]
? vecextract(v, "^2") \\ complement
%5 = [a, c, d, e]
? vecextract(matid(3), "2..", "..")
%6 =
[0 1 0]
[0 0 1]
```

The range notations v[i..j] and $v[^i]$ (for t_VEC or t_COL) and M[i..j, k..1] and friends (for t_MAT) implement a subset of the above, in a simpler and *faster* way, hence should be preferred in most common situations. The following features are not implemented in the range notation:

- · reverse order,
- omitting either a or b in :math:`a..:math:b`.

vecmax(v)

If x is a vector or a matrix, returns the largest entry of x, otherwise returns a copy of x. Error if x is empty.

If v is given, set it to the index of a largest entry (indirect maximum), when x is a vector. If x is a matrix, set v to coordinates [i, j] such that x[i, j] is a largest entry. This flag is ignored if x is not a vector or matrix.

? vecmax([10, 20, -30, 40]) %1 = 40

? vecmax([10, 20, -30, 40], &v); v
%2 = 4
? vecmax([10, 20; -30, 40], &v); v
%3 = [2, 2]

vecmin(v)

If x is a vector or a matrix, returns the smallest entry of x, otherwise returns a copy of x. Error if x is empty.

If v is given, set it to the index of a smallest entry (indirect minimum), when x is a vector. If x is a matrix, set v to coordinates [i, j] such that x[i, j] is a smallest entry. This is ignored if x is not a vector or matrix.

```
? vecmin([10, 20, -30, 40])
%1 = -30
? vecmin([10, 20, -30, 40], &v); v
%2 = 3
? vecmin([10, 20; -30, 40], &v); v
%3 = [2, 1]
```

vecprod()

Return the product of the components of the vector v. Return 1 on an empty vector.

? vecprod([1,2,3])
%1 = 6
? vecprod([])
%2 = 1

vecsearch(x, cmpf)

Determines whether x belongs to the sorted vector or list v: return the (positive) index where x was found, or 0 if it does not belong to v.

If the comparison function cmpf is omitted, we assume that v is sorted in increasing order, according to the standard comparison function lex, thereby restricting the possible types for x and the elements of v (integers, fractions, reals, and vectors of such). We also transparently allow a t_VECSMALL x in this case, for the natural ordering of the integers.

If cmpf is present, it is understood as a comparison function and we assume that v is sorted according to it, see vecsort for how to encode comparison functions.

```
? v = [1,3,4,5,7];
? vecsearch(v, 3)
%2 = 2
? vecsearch(v, 6)
%3 = 0 \\ not in the list
? vecsearch([7,6,5], 5) \\ unsorted vector: result undefined
%4 = 0
```

Note that if we are sorting with respect to a key which is expensive to compute (e.g. a discriminant), one should rather precompute all keys, sort that vector and search in the vector of keys, rather than searching in the original vector with respect to a comparison function.

By abuse of notation, x is also allowed to be a matrix, seen as a vector of its columns; again by abuse of notation, a t_VEC is considered as part of the matrix, if its transpose is one of the matrix columns.
```
? v = vecsort([3,0,2; 1,0,2]) \\ sort matrix columns according to lex order
%1 =
[0 2 3]
[0 2 1]
? vecsearch(v, [3,1]~)
%2 = 3
? vecsearch(v, [3,1]) \\ can search for x or x~
%3 = 3
? vecsearch(v, [1,2])
%4 = 0 \\ not in the list
```

vecsort(cmpf, flag)

Sorts the vector x in ascending order, using a mergesort method. x must be a list, vector or matrix (seen as a vector of its columns). Note that mergesort is stable, hence the initial ordering of "equal" entries (with respect to the sorting criterion) is not changed.

If cmpf is omitted, we use the standard comparison function lex, thereby restricting the possible types for the elements of x (integers, fractions or reals and vectors of those). We also transparently allow a t_VECSMALL x in this case, for the standard ordering on the integers.

If cmpf is present, it is understood as a comparison function and we sort according to it. The following possibilities exist:

- an integer k: sort according to the value of the k-th subcomponents of the components of x.
- a vector: sort lexicographically according to the components listed in the vector. For example, if cmpf = [2, 1, 3], sort with respect to the second component, and when these are equal, with respect to the first, and when these are equal, with respect to the third.
- a comparison function: t_CLOSURE with two arguments x and y, and returning a real number which is < 0,
 > 0 or = 0 if x < y, x > y or x = y respectively.
- a key: t_CLOSURE with one argument x and returning the value f(x) with respect to which we sort.

```
? vecsort([3,0,2; 1,0,2]) \\ sort columns according to lex order
%1 =
[0 2 3]
[0 2 1]
? vecsort(v, (x,y)->y-x) \\ reverse sort
? vecsort(v, (x,y)->abs(x)-abs(y)) \\ sort by increasing absolute value
? vecsort(v, abs) \\ sort by increasing absolute value, using key
? cmpf(x,y) = my(dx = poldisc(x), dy = poldisc(y)); abs(dx) - abs(dy);
? v = [x^2+1, x^3-2, x^4+5*x+1] vecsort(v, cmpf) \\ comparison function
? vecsort(v, x->abs(poldisc(x))) \\ key
```

The abs and cmpf examples show how to use a named function instead of an anonymous function. It is preferable to use a *key* whenever possible rather than include it in the comparison function as above since the key is evaluated O(n) times instead of $O(n \log n)$, where n is the number of entries.

A direct approach is also possible and equivalent to using a sorting key:

```
? T = [abs(poldisc(x)) | x<-v];
? perm = vecsort(T,,1); \\ indirect sort
? vecextract(v, perm)
```

This also provides the vector T of all keys, which is interesting for instance in later vecsearch calls: it is more efficient to sort T (T = vecextract(T, perm)) then search for a key in T rather than to search in v using a comparison function or a key. Note also that mapisdefined is often easier to use and faster than vecsearch.

The binary digits of *flag* mean:

- 1: indirect sorting of the vector x, i.e. if x is an n-component vector, returns a permutation of [1, 2, ..., n] which applied to the components of x sorts x in increasing order. For example, vecextract(x, vecsort(x,,1)) is equivalent to vecsort(x).
- 4: use descending instead of ascending order.
- 8: remove "duplicate" entries with respect to the sorting function (keep the first occurring entry). For example:

```
? vecsort([Pi,Mod(1,2),z], (x,y)->0, 8) \\ make everything compare equal
%1 = [3.141592653589793238462643383]
? vecsort([[2,3],[0,1],[0,3]], 2, 8)
%2 = [[0, 1], [2, 3]]
```

vecsum()

Return the sum of the components of the vector v. Return 0 on an empty vector.

? vecsum([1,2,3])
%1 = 6
? vecsum([])
%2 = 0

weber(flag, precision)

One of Weber's three f functions. If flag = 0, returns

 $f(x) = \exp(-i\pi/24) \cdot \eta((x+1)/2) / \eta(x) such that j = (f^{24} - 16)^3 / f^{24},$

where j is the elliptic j-invariant (see the function ellj). If flag = 1, returns

 $f_1(x) = \eta(x/2)/\eta(x) such that j = (f_1^{24} + 16)^3/f_1^{24}.$

Finally, if flag = 2, returns

 $f_2(x) = \sqrt{2\eta(2x)}/\eta(x) such that j = (f_2^{24} + 16)^3/f_2^{24}.$

Note the identities $f^8 = f_1^8 + f_2^8$ and $ff_1f_2 = \sqrt{2}$.

zeta(precision)

For s! = 1 a complex number, Riemann's zeta function $\zeta(s) = \sum_{n>=1} n^{-s}$, computed using the Euler-Maclaurin summation formula, except when s is of type integer, in which case it is computed using Bernoulli numbers for s <= 0 or s > 0 and even, and using modular forms for s > 0 and odd. Power series are also allowed:

For s! = 1 a *p*-adic number, Kubota-Leopoldt zeta function at *s*, that is the unique continuous *p*-adic function on the *p*-adic integers that interpolates the values of $(1 - p^{-k})\zeta(k)$ at negative integers *k* such that k = 1(modp - 1) (resp. *k* is odd) if *p* is odd (resp. p = 2). Power series are not allowed in this case.

zetahurwitz(*x*, *der*, *precision*)

Hurwitz zeta function $\zeta(s, x) = \sum_{n>=0} (n+x)^{-s}$ and analytically continued, with s! = 1 and x not a negative or zero integer. Note that $\zeta(s, 1) = \zeta(s)$. s can also be a polynomial, rational function, or power series. If der is positive, compute the der'th derivative with respect to s. Note that the derivative with respect to x is simply $-s\zeta(s+1,x)$.

```
? zetahurwitz(Pi,Pi)
%1 = 0.056155444497585099925180502385781494484
? zetahurwitz(2,1) - zeta(2)
%2 = -2.350988701644575016 E-38
? zetahurwitz(Pi,3) - (zeta(Pi)-1-1/2^Pi)
%3 = -2.2040519077917890774 E-39
? zetahurwitz(-7/2,1) - zeta(-7/2)
\%4 = -2.295887403949780289 E-41
? zetahurwitz(-2.3,Pi+I*log(2))
%5 = -5.1928369229555125820137832704455696057\
- 6.1349660138824147237884128986232049582*I
? zetahurwitz(-1+x^2+0(x^3),1)
- 0.16542114370045092921391966024278064276*x^2 + 0(x^3)
? zetahurwitz(1+x+0(x^4),2)
- 0.42278433509846713939348790991759756896
+ 0.072815845483676724860586375874901319138*x + 0(x^2)
? zetahurwitz(2,1,2) \setminus zeta''(2)
\%8 = 1.9892802342989010234208586874215163815
```

zetamult(t, precision)

For s a vector of positive integers such that $s[1] \ge 2$, returns the multiple zeta value (MZV)

$$\zeta(s_1, ..., s_k) = \sum_{n_1 > ... > n_k > 0} n_1^{-s_1} ... n_k^{-s_k}$$

of length k and weight $\sum_i s_i$. More generally, return Yamamoto's t-MZV interpolation evaluated at t: for t = 0, this is the ordinary MZV; for t = 1, we obtain the MZSV star value, with $\geq =$ instead of strict inequalities; and of course, for t = t' we obtain Yamamoto's one-variable polynomial.

```
? zetamult([2,1]) - zeta(3) \\ Euler's identity
%1 = 0.E-38
? zetamult([2,1], 1) \\ star value
%2 = 2.4041138063191885707994763230228999815
? zetamult([2,1], 'x)
%3 = 1.20205[...]*x + 1.20205[...]
```

If the bit precision is B, this function runs in time $O(k(B+k)^2)$ if t = 0, and $O(kB^3)$ otherwise.

In addition to the above format (avec), the function also accepts a binary word format evec (each s_i is replaced by s_i bits, all of them 0 but the last one) giving the MZV representation as an iterated integral, and an index format (if e is the positive integer attached the evec vector of bits, the index is the integer $e + 2^{k-2}$). The function zetamultconvert allows to pass from one format to the other; the function zetamultall computes simultaneously all MZVs of weight $\sum_{i < =k} s_i$ up to n.

zetamultconvert(fl)

a being either an evec, avec, or index m, converts into evec (fl = 0), avec (fl = 1), or index m (fl = 2).

```
? zetamultconvert(10)
%1 = Vecsmall([3, 2])
? zetamultconvert(13)
%2 = Vecsmall([2, 2, 1])
? zetamultconvert(10, 0)
%3 = Vecsmall([0, 0, 1, 0, 1])
? zetamultconvert(13, 0)
%4 = Vecsmall([0, 1, 0, 1, 1])
```

The last two lines imply that [3, 2] and [2, 2, 1] are dual (reverse order of bits and swap 0 and 1 in evec form). Hence they have the same zeta value:

```
? zetamult([3,2])
%5 = 0.22881039760335375976874614894168879193
? zetamult([2,2,1])
%6 = 0.22881039760335375976874614894168879193
```

zetamultdual()

s being either an evec, avec, or index m, return the dual sequence in avec format. The dual of a sequence of length r and weight k has length k - r and weight k. Duality is an involution and zeta values attached to dual sequences are the same:

? zetamultdual([4])
%1 = Vecsmall([2, 1, 1])
? zetamultdual(%)
%2 = Vecsmall([4])
? zetamult(%1) - zetamult(%2)
%3 = 0.E-38

In evec form, duality simply reverses the order of bits and swaps 0 and 1:

```
? zetamultconvert([4], 0)
%4 = Vecsmall([0, 0, 0, 1])
? zetamultconvert([2,1,1], 0)
%5 = Vecsmall([0, 1, 1, 1])
```

znchar()

Given a datum D describing a group $(\mathbb{Z}/N\mathbb{Z})^*$ and a Dirichlet character χ , return the pair [G, chi], where G is znstar(N, 1)) and chi is a GP character.

The following possibilities for D are supported

- a nonzero t_INT congruent to 0, 1 modulo 4, return the real character modulo D given by the Kronecker symbol (D/.);
- a t_INTMOD Mod(m, N), return the Conrey character modulo N of index m (see znconreylog).

• a modular form space as per $mfinit([N, k, \chi])$ or a modular form for such a space, return the underlying Dirichlet character χ (which may be defined modulo a divisor of N but need not be primitive).

In the remaining cases, G is initialized by znstar(N, 1).

- a pair [G, chi], where chi is a standard GP Dirichlet character c = (c_j) on G (generic character t_VEC or Conrey characters t_COL or t_INT); given generators G = ⊕(ℤ/d_jℤ)g_j, χ(g_j) = e(c_j/d_j).
- a pair [G, chin], where chin is a *normalized* representation [n, c] of the Dirichlet character c; $\chi(g_j) = e(c_j/n)$ where n is minimal (order of χ).

```
? [G,chi] = znchar(-3);
? G.cyc
%2 = [2]
? chareval(G, chi, 2)
%3 = 1/2
? kronecker(-3,2)
%4 = -1
? znchartokronecker(G,chi)
%5 = -3
? mf = mfinit([28, 5/2, Mod(2,7)]); [f] = mfbasis(mf);
? [G,chi] = znchar(mf); [G.mod, chi]
%7 = [7, [2]~]
? [G,chi] = znchar(f); chi
%8 = [28, [0, 2]~]
```

zncharconductor(chi)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$ (see dirichletchar (in the PARI manual) or ??character). Return the conductor of chi:

```
? G = znstar(126000, 1);
? zncharconductor(G,11) \\ primitive
%2 = 126000
? zncharconductor(G,1) \\ trivial character, not primitive!
%3 = 1
? zncharconductor(G,1009) \\ character mod 5^3
%4 = 125
```

znchardecompose(chi, Q)

Let $N = \prod_p p^{e_p}$ and a Dirichlet character χ , we have a decomposition $\chi = \prod_p \chi_p$ into character modulo N where the conductor of χ_p divides p^{e_p} ; it equals p^{e_p} for all p if and only if χ is primitive.

Given a *znstar* G describing a group $(\mathbb{Z}/N\mathbb{Z})^*$, a Dirichlet character chi and an integer Q, return $\prod_{p \parallel (Q,N)} \chi_p$. For instance, if Q = p is a prime divisor of N, the function returns χ_p (as a character modulo N), given as a Conrey character (t_COL).

```
? G = znstar(40, 1);
? G.cyc
%2 = [4, 2, 2]
? chi = [2, 1, 1];
? chi2 = znchardecompose(G, chi, 2)
%4 = [1, 1, 0]~
? chi5 = znchardecompose(G, chi, 5)
%5 = [0, 0, 2]~
? znchardecompose(G, chi, 3)
```

%6 = [0, 0, 0]~ ? c = charmul(G, chi2, chi5) %7 = [1, 1, 2]~ \\ t_COL: in terms of Conrey generators ! ? znconreychar(G,c) %8 = [2, 1, 1] \\ t_VEC: in terms of SNF generators

znchargauss(chi, a, precision)

Given a Dirichlet character χ on $G = (\mathbb{Z}/N\mathbb{Z})^*$ (see znchar), return the complex Gauss sum

$$g(\chi, a) = \sum_{n=1}^{N} \chi(n) e(an/N)$$

? [G,chi] = znchar(-3); \\ quadratic Gauss sum: I*sqrt(3) ? znchargauss(G,chi) %2 = 1.7320508075688772935274463415058723670*I ? [G,chi] = znchar(5); ? znchargauss(G,chi) \\ sqrt(5) %2 = 2.2360679774997896964091736687312762354 ? G = znstar(300,1); chi = [1,1,12]~; ? znchargauss(G,chi) / sqrt(300) - exp(2*I*Pi*11/25) \\ = 0 %4 = 2.350988701644575016 E-38 + 1.4693679385278593850 E-39*I ? lfuntheta([G,chi], 1) \\ = 0 %5 = -5.79[...] E-39 - 2.71[...] E-40*I

zncharinduce(chi, N)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Let N be a multiple of q, return the character modulo N extending chi. As usual for arithmetic functions, the new modulus N can be given as a t_INT, via a factorization matrix or a pair [N, factor(N)], or by znstar(N, 1).

```
? G = znstar(4, 1);
? chi = znconreylog(G,1); \\ trivial character mod 4
? zncharinduce(G, chi, 80) \\ now mod 80
\%3 = [0, 0, 0] \sim
? zncharinduce(G, 1, 80) \\ same using directly Conrey label
\%4 = [0, 0, 0] \sim
? G2 = znstar(80, 1);
? zncharinduce(G, 1, G2) \\ same
\%4 = [0, 0, 0] \sim
? chi = zncharinduce(G, 3, G2) \setminus extend the nontrivial character mod 4
\%5 = [1, 0, 0] \sim
? [G0,chi0] = znchartoprimitive(G2, chi);
? G0.mod
\%7 = 4
? chi0
%8 = [1]~
```

Here is a larger example:

```
? G = znstar(126000, 1);
? label = 1009;
? chi = znconreylog(G, label)
%3 = [0, 0, 0, 14, 0]~
? [G0,chi0] = znchartoprimitive(G, label); \\ works also with 'chi'
? G0.mod
%5 = 125
? chi0 \\ primitive character mod 5^3 attached to chi
%6 = [14]~
? G0 = znstar(N0, 1);
? zncharinduce(G0, chi0, G) \\ induce back
%8 = [0, 0, 0, 14, 0]~
? znconreyexp(G, %)
%9 = 1009
```

zncharisodd(chi)

Let G be attached to $(\mathbb{Z}/N\mathbb{Z})^*$ (as per G = znstar(N,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/N\mathbb{Z})^*$, given by

- a t_VEC: a standard character on G.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Return 1 if and only if chi(-1) = -1 and 0 otherwise.

```
? G = znstar(8, 1);
? zncharisodd(G, 1) \\ trivial character
%2 = 0
? zncharisodd(G, 3)
%3 = 1
? chareval(G, 3, -1)
%4 = 1/2
```

znchartokronecker(chi, flag)

Let G be attached to $(\mathbb{Z}/N\mathbb{Z})^*$ (as per G = znstar(N,1)) and let chi be a Dirichlet character on $(\mathbb{Z}/N\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

If flag = 0, return the discriminant D if chi is real equal to the Kronecker symbol (D/.) and 0 otherwise. The discriminant D is fundamental if and only if chi is primitive.

If flag = 1, return the fundamental discriminant attached to the corresponding primitive character.

```
? G = znstar(8,1); CHARS = [1,3,5,7]; \\ Conrey labels
? apply(t->znchartokronecker(G,t), CHARS)
%2 = [4, -8, 8, -4]
? apply(t->znchartokronecker(G,t,1), CHARS)
%3 = [1, -8, 8, -4]
```

znchartoprimitive(chi)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, of conductor $q_0 || q$.

```
? G = znstar(126000, 1);
? [G0,chi0] = znchartoprimitive(G,11)
? G0.mod
\%3 = 126000
? chi0
\%4 = 11
? [G0,chi0] = znchartoprimitive(G,1);\\ trivial character, not primitive!
? G0.mod
\%6 = 1
? chi0
%7 = []~
? [G0,chi0] = znchartoprimitive(G,1009)
? G0.mod
\%4 = 125
? chi0
%5 = [14]~
```

Note that znconreyconductor is more efficient since it can return χ_0 and its conductor q_0 without needing to initialize G_0 . The price to pay is a more cryptic format and the need to initialize G_0 later, but that needs to be done only once for all characters with conductor q_0 .

znconreychar(m)

Given a *znstar* G attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), this function returns the Dirichlet character attached to $m \in (\mathbb{Z}/q\mathbb{Z})^*$ via Conrey's logarithm, which establishes a "canonical" bijection between $(\mathbb{Z}/q\mathbb{Z})^*$ and its dual.

Let $q = \prod_p p^{e_p}$ be the factorization of q into distinct primes. For all odd p with $e_p > 0$, let g_p be the element in $(\mathbb{Z}/q\mathbb{Z})^*$ which is

- congruent to $1 \mod q/p^{e_p}$,
- congruent mod p^{e_p} to the smallest positive integer that generates $(\mathbb{Z}/p^2\mathbb{Z})^*$.

For p = 2, we let g_4 (if $2^{e_2} \ge 4$) and g_8 (if furthermore $(2^{e_2} \ge 8)$ be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to $1 \mod q/2^{e_2}$,
- $g_4 = -1 \mod 2^{e_2}$,
- $g_8 = 5mod2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} \ge 2$) are independent generators of $(\mathbb{Z}/q\mathbb{Z})^*$, i.e. every m in $(\mathbb{Z}/q\mathbb{Z})^*$ can be written uniquely as $\prod_p g_p^{m_p}$, where m_p is defined modulo the order o_p of g_p and $p \in S_q$, the set of prime divisors of q together with 4 if 4 || q and 8 if 8 || q. Note that the g_p are in general *not* SNF generators as produced by znstar whenever $\omega(q) \ge 2$, although their number is the same. They however allow to handle the finite abelian group $(\mathbb{Z}/q\mathbb{Z})^*$ in a fast and elegant way. (Which unfortunately does not generalize to ray class groups or Hecke characters.)

The Conrey logarithm of m is the vector $(m_p)_{p \in S_q}$, obtained via znconreylog. The Conrey character $\chi_q(m, .)$ attached to $m \mod q$ maps each g_p , $p \in S_q$ to $e(m_p/o_p)$, where $e(x) = \exp(2i\pi x)$. This function returns the Conrey character expressed in the standard PARI way in terms of the SNF generators G.gen.

```
? G = znstar(8,1);
? G.cyc
```

 $\%2 = [2, 2] \setminus Z/2 \times Z/2$? G.gen %3 = [7, 3]? $znconreychar(G,1) \setminus 1$ is always the trivial character %4 = [0, 0]? znconreychar(G,2) \setminus 2 is not coprime to 8 !!! *** at top-level: znconreychar(G,2) *** znconreychar: elements not coprime in Zideallog: 2 8 *** Break loop: type 'break' to go back to GP prompt break> ? znconreychar(G,3) %5 = [0.1]? znconreychar(G,5) %6 = [1, 1]? znconreychar(G,7) %7 = [1, 0]

We indeed get all 4 characters of $(\mathbb{Z}/8\mathbb{Z})^*$.

For convenience, we allow to input the *Conrey logarithm* of *m* instead of *m*:

? G = znstar(55, 1); ? znconreychar(G,7) %2 = [7, 0] ? znconreychar(G, znconreylog(G,7)) %3 = [7, 0]

znconreyconductor(chi, chi0)

Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)) and chi be a Dirichlet character on $(\mathbb{Z}/q\mathbb{Z})^*$, given by

- a t_VEC: a standard character on bid.gen,
- a t_INT or a t_COL: a Conrey index in (ℤ/qℤ)* or its Conrey logarithm; see dirichletchar (in the PARI manual) or ??character.

Return the conductor of chi, as the t_INT bid.mod if chi is primitive, and as a pair [N, faN] (with faN the factorization of N) otherwise.

If chi0 is present, set it to the Conrey logarithm of the attached primitive character.

```
? G = znstar(126000, 1);
? znconreyconductor(G,11) \\ primitive
%2 = 126000
? znconreyconductor(G,1) \\ trivial character, not primitive!
%3 = [1, matrix(0,2)]
? N0 = znconreyconductor(G,1009, &chi0) \\ character mod 5^3
%4 = [125, Mat([5, 3])]
? chi0
%5 = [14]~
? G0 = znstar(N0, 1); \\ format [N,factor(N)] accepted
```

```
? znconreyexp(G0, chi0)
%7 = 9
? znconreyconductor(G0, chi0) \\ now primitive, as expected
%8 = 125
```

The group GO is not computed as part of znconreyconductor because it needs to be computed only once per conductor, not once per character.

znconreyexp(chi)

Given a *znstar* G attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q, 1)), this function returns the Conrey exponential of the character *chi*: it returns the integer $m \in (\mathbb{Z}/q\mathbb{Z})^*$ such that znconreylog(G, :math:`m)` is *chi*.

The character chi is given either as a

- t_VEC: in terms of the generators G.gen;
- t_COL: a Conrey logarithm.

```
? G = znstar(126000, 1)
? znconreylog(G,1)
%2 = [0, 0, 0, 0, 0]~
? znconreyexp(G,%)
\%3 = 1
? G.cyc \\ SNF generators
\%4 = [300, 12, 2, 2, 2]
? chi = [100, 1, 0, 1, 0]; \\ some random character on SNF generators
? znconreylog(G, chi) \\ in terms of Conrey generators
\%6 = [0, 3, 3, 0, 2] \sim
? znconreyexp(G, %) \\ apply to a Conrey log
\%7 = 18251
? znconreyexp(G, chi) \\ ... or a char on SNF generators
\%8 = 18251
? znconreychar(G,%)
\%9 = [100, 1, 0, 1, 0]
```

znconreylog(m)

Given a *znstar* attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per G = znstar(q,1)), this function returns the Conrey logarithm of $m \in (\mathbb{Z}/q\mathbb{Z})^*$.

Let $q = \prod_p p^{e_p}$ be the factorization of q into distinct primes, where we assume $e_2 = 0$ or $e_2 \ge 2$. (If $e_2 = 1$, we can ignore 2 from the factorization, as if we replaced q by q/2, since $(\mathbb{Z}/q\mathbb{Z})^* (\mathbb{Z}/(q/2)\mathbb{Z})^*$.)

For all odd p with $e_p > 0$, let g_p be the element in $(\mathbb{Z}/q\mathbb{Z})^*$ which is

- congruent to $1 \mod q/p^{e_p}$,
- congruent mod p^{e_p} to the smallest positive integer that generates $(\mathbb{Z}/p^2\mathbb{Z})^*$.

For p = 2, we let g_4 (if $2^{e_2} \ge 4$) and g_8 (if furthermore ($2^{e_2} \ge 8$) be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to $1 \mod q/2^{e_2}$,
- $g_4 = -1 \mod 2^{e_2}$,
- $g_8 = 5mod2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} \ge 2$) are independent generators of $\mathbb{Z}/q\mathbb{Z}^*$, i.e. every m in $(\mathbb{Z}/q\mathbb{Z})^*$ can be written uniquely as $\prod_p g_p^{m_p}$, where m_p is defined modulo the order o_p of g_p and $p \in S_q$, the set of prime divisors of q together with 4 if 4 || q and 8 if 8 || q. Note that the g_p are in general *not* SNF generators as produced

by znstar whenever $\omega(q) >= 2$, although their number is the same. They however allow to handle the finite abelian group $(\mathbb{Z}/q\mathbb{Z})^*$ in a fast and elegant way. (Which unfortunately does not generalize to ray class groups or Hecke characters.)

The Conrey logarithm of m is the vector $(m_p)_{p \in S_q}$. The inverse function znconreyexp recovers the Conrey label m from a character.

```
? G = znstar(126000, 1);
? znconreylog(G,1)
\%2 = [0, 0, 0, 0, 0] \sim
? znconreyexp(G, %)
\%3 = 1
? znconreylog(G,2) \\ 2 is not coprime to modulus !!!
 *** at top-level: znconreylog(G,2)
 *** ^_____
 *** znconreylog: elements not coprime in Zideallog:
 2
 126000
*** Break loop: type 'break' to go back to GP prompt
break>
? znconreylog(G,11) \\ wrt. Conrey generators
\%4 = [0, 3, 1, 76, 4] \sim
? log11 = ideallog(,11,G) \\ wrt. SNF generators
%5 = [178, 3, -75, 1, 0]~
```

For convenience, we allow to input the ordinary discrete log of m, ideallog(, m, bid), which allows to convert discrete logs from bid.gen generators to Conrey generators.

```
? znconreylog(G, log11)
%7 = [0, 3, 1, 76, 4]~
```

We also allow a character (t_VEC) on bid.gen and return its representation on the Conrey generators.

```
? G.cyc
%8 = [300, 12, 2, 2, 2]
? chi = [10,1,0,1,1];
? znconreylog(G, chi)
%10 = [1, 3, 3, 10, 2]~
? n = znconreyexp(G, chi)
%11 = 84149
? znconreychar(G, n)
%12 = [10, 1, 0, 1, 1]
```

zncoppersmith(N, X, B)

Coppersmith's algorithm. N being an integer and $P \in \mathbb{Z}[t]$, finds in polynomial time in $\log(N)$ and d = deg(P) all integers x with $||x|| \le X$ such that

$$gcd(N, P(x)) >= B.$$

This is a famous application of the LLL algorithm meant to help in the factorization of N. Notice that P may be reduced modulo $N\mathbb{Z}[t]$ without affecting the situation. The parameter X must not be too large: assume for now that the leading coefficient of P is coprime to N, then we must have

$$d\log X\log N < \log^2 B,$$

i.e., $X < N^{1/d}$ when B = N. Let now P_0 be the gcd of the leading coefficient of P and N. In applications to factorization, we should have $P_0 = 1$; otherwise, either $P_0 = N$ and we can reduce the degree of P, or P_0 is a non trivial factor of N. For completeness, we nevertheless document the exact conditions that X must satisfy in this case: let $p := \log_N P_0$, $b := \log_N B$, $x := \log_N X$, then

- either $p \ge d/(2d-1)$ is large and we must have xd < 2b-1;
- or p < d/(2d-1) and we must have both p < b < 1 p + p/d and $x(d + p(1 2d)) < (b p)^2$. Note that this reduces to $xd < b^2$ when p = 0, i.e., the condition described above.

Some x larger than X may be returned if you are very lucky. The routine runs in polynomial time in $\log N$ and d but the smaller B, or the larger X, the slower. The strength of Coppersmith method is the ability to find roots modulo a general *composite* N: if N is a prime or a prime power, polrootsmod or polrootspadic will be much faster.

We shall now present two simple applications. The first one is finding nontrivial factors of N, given some partial information on the factors; in that case B must obviously be smaller than the largest nontrivial divisor of N.

```
setrand(1); \\ to make the example reproducible
[a,b] = [10^30, 10^31]; D = 20;
p = randomprime([a,b]);
q = randomprime([a,b]); N = p*q;
\\ assume we know 0) p | N; 1) p in [a,b]; 2) the last D digits of p
p0 = p % 10^D;
? L = zncoppersmith(10^D*x + p0, N, b \ 10^D, a)
time = 1ms.
%6 = [738281386540]
? gcd(L[1] * 10^D + p0, N) == p
%7 = 1
```

and we recovered p, faster than by trying all possibilities $x < 10^{11}$.

The second application is an attack on RSA with low exponent, when the message x is short and the padding P is known to the attacker. We use the same RSA modulus N as in the first example:

```
setrand(1);
P = random(N); \\ known padding
e = 3; \\ small public encryption exponent
X = floor(N^0.3); \\ N^(1/e - epsilon)
x0 = random(X); \\ unknown short message
C = lift( (Mod(x0,N) + P)^e ); \\ known ciphertext, with padding P
zncoppersmith((P + x)^3 - C, N, X)
\\ result in 244ms.
%14 = [2679982004001230401]
? %[1] == x0
%15 = 1
```

We guessed an integer of the order of 10^{18} , almost instantly.

$\mathbf{znlog}(g, o)$

This functions allows two distinct modes of operation depending on g:

• if g is the output of znstar (with initialization), we compute the discrete logarithm of x with respect to the generators contained in the structure. See ideallog for details.

else g is an explicit element in (Z/NZ)*, we compute the discrete logarithm of x in (Z/NZ)* in base g. The
rest of this entry describes the latter possibility.

The result is [] when x is not a power of g, though the function may also enter an infinite loop in this case.

If present, o represents the multiplicative order of g, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord is the order of g. This provides a definite speedup when the discrete log problem is simple:

```
? p = nextprime(10^4); g = znprimroot(p); o = [p-1, factor(p-1)];
? for(i=1,10^4, znlog(i, g, o))
time = 163 ms.
? for(i=1,10^4, znlog(i, g))
time = 200 ms. \\ a little slower
```

The result is undefined if g is not invertible mod N or if the supplied order is incorrect.

This function uses

- a combination of generic discrete log algorithms (see below).
- in (ℤ/Nℤ)* when N is prime: a linear sieve index calculus method, suitable for N < 10⁵⁰, say, is used for large prime divisors of the order.

The generic discrete log algorithms are:

- Pohlig-Hellman algorithm, to reduce to groups of prime order q, where q || p 1 and p is an odd prime divisor of N,
- Shanks baby-step/giant-step ($q < 2^{32}$ is small),
- Pollard rho method $(q > 2^{32})$.

The latter two algorithms require $O(\sqrt{q})$ operations in the group on average, hence will not be able to treat cases where $q > 10^{30}$, say. In addition, Pollard rho is not able to handle the case where there are no solutions: it will enter an infinite loop.

```
? g = znprimroot(101)
\%1 = Mod(2, 101)
? znlog(5, g)
%2 = 24
? g^24
\%3 = Mod(5, 101)
? G = znprimroot(2 * 101^{10})
%4 = Mod(110462212541120451003, 220924425082240902002)
? znlog(5, G)
%5 = 76210072736547066624
? G^% == 5
\%6 = 1
? N = 2^4*3^2*5^3*7^4*11; g = Mod(13, N); znlog(g^110, g)
\%7 = 110
? znlog(6, Mod(2,3)) \setminus no solution
%8 = []
```

For convenience, g is also allowed to be a p-adic number:

```
? g = 3+0(5^10); znlog(2, g)
%1 = 1015243
```

? g[%] %2 = 2 + O(5¹0)

znorder(o)

x must be an integer mod n, and the result is the order of x in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. Returns an error if x is not invertible. The parameter o, if present, represents a nonzero multiple of the order of x, see DLfun (in the PARI manual); the preferred format for this parameter is [ord, factor(ord)], where ord = eulerphi(n) is the cardinality of the group.

znprimroot()

Returns a primitive root (generator) of $(\mathbb{Z}/n\mathbb{Z})^*$, whenever this latter group is cyclic (n = 4 or $n = 2p^k$ or $n = p^k$, where p is an odd prime and $k \ge 0$). If the group is not cyclic, the result is undefined. If n is a prime power, then the smallest positive primitive root is returned. This may not be true for $n = 2p^k$, p odd.

Note that this function requires factoring p-1 for p as above, in order to determine the exact order of elements in $(\mathbb{Z}/n\mathbb{Z})^*$: this is likely to be costly if p is large.

znstar(flag)

Gives the structure of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. The output G depends on the value of *flag*:

• flag = 0 (default), an abelian group structure [h, d, g], where $h = \phi(n)$ is the order (G.no), d (G.cyc) is a k-component row-vector d of integers d_i such that $d_i > 1$, $d_i || d_{i-1}$ for $i \ge 2$ and

and:math: g'(:literal: G.gen') is a: math: k'-component row vector giving generators of the image of the cyclic group of the

• flag = 1 the result is a bid structure; this allows computing discrete logarithms using znlog (also in the noncyclic case!).

```
? G = znstar(40)
%1 = [16, [4, 2, 2], [Mod(17, 40), Mod(21, 40), Mod(11, 40)]]
? G.no \\ eulerphi(40)
%2 = 16
? G.cyc \\ cycle structure
%3 = [4, 2, 2]
? G.gen \\ generators for the cyclic components
%4 = [Mod(17, 40), Mod(21, 40), Mod(11, 40)]
? apply(znorder, G.gen)
%5 = [4, 2, 2]
```

For user convenience, we define znstar(0) as [2, [2], [-1]], corresponding to \mathbb{Z}^* , but flag = 1 is not implemented in this trivial case.

cypari2.gen.objtogen(s)

Convert any SageMath/Python object to a PARI Gen.

For SageMath types, this uses the __pari__() method on the object. Basic Python types like int are converted directly. For other types, the string representation is used.

Examples:

```
>>> from cypari2 import Pari
>>> pari = Pari()
```

>>> pari(0)
0
>>> pari([2,3,5])
[2, 3, 5]

```
>>> a = pari(1)
>>> a, a.type()
(1, 't_INT')
```

```
>>> from fractions import Fraction
>>> a = pari(Fraction('1/2'))
>>> a, a.type()
(1/2, 't_FRAC')
```

Conversion from reals uses the real's own precision:

```
>>> a = pari(1.2); a, a.type(), a.bitprecision()
(1.2000000000000, 't_REAL', 64)
```

Conversion from strings uses the current PARI real precision. By default, this is 64 bits:

```
>>> a = pari('1.2'); a, a.type(), a.bitprecision()
(1.2000000000000, 't_REAL', 64)
```

Unicode and bytes work fine:

```
>>> pari(b"zeta(3)")
1.20205690315959
>>> pari(u"zeta(3)")
1.20205690315959
```

But we can change this precision:

```
>>> pari.set_real_precision(35) # precision in decimal digits
15
>>> a = pari('Pi'); a, a.type(), a.bitprecision()
(3.1415926535897932384626433832795029, 't_REAL', 128)
>>> a = pari('1.2'); a, a.type(), a.bitprecision()
(1.20000000000000000000000000000000, 't_REAL', 128)
```

Set the precision to 15 digits for the remaining tests:

```
>>> pari.set_real_precision(15)
35
```

Conversion from basic Python types:

```
>>> pari(int(-5))
-5
>>> pari(2**150)
1427247692705959881058285969449495136382746624
>>> import math
>>> pari(math.pi)
```

```
3.14159265358979
>>> one = pari(complex(1,0)); one, one.type()
(1.0000000000000, 't_COMPLEX')
>>> pari(complex(0, 1))
1.00000000000000*I
>>> pari(complex(0.3, 1.7))
0.30000000000000 + 1.700000000000*I
```

```
>>> pari(False)
0
>>> pari(True)
1
```

The following looks strange, but it is what PARI does:

```
>>> pari(["print(x)"])
x
[0]
>>> pari("[print(x)]")
x
[0]
```

Tests:

```
>>> pari(None)
Traceback (most recent call last):
....
ValueError: Cannot convert None to pari
```

CHAPTER

THREE

MEMORY MANAGEMENT FOR GENS ON THE PARI STACK OR THE HEAP

class cypari2.stack.DetachGen

Destroy a Gen but keep the GEN which is inside it.

The typical usage is as follows:

- 1. Creates the DetachGen object from a :class`Gen`.
- 2. Removes all other references to that Gen.
- 3. Call the detach method to retrieve the GEN (or a copy of it if the original was not on the stack).

CHAPTER

FOUR

CONVERT PYTHON FUNCTIONS TO PARI CLOSURES

AUTHORS:

• Jeroen Demeyer (2015-04-10): initial version, Sage ticket #18052.

Examples:

```
>>> def the_answer():
... return 42
>>> import cypari2
>>> pari = cypari2.Pari()
>>> f = pari(the_answer)
>>> f()
42
```

```
>>> cube = pari(lambda i: i**3)
>>> cube.apply(range(10))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

cypari2.closure.objtoclosure(f)

Convert a Python function (more generally, any callable) to a PARI t_CLOSURE.

Note: With the current implementation, the function can be called with at most 5 arguments.

Warning: The function **f** which is called through the closure cannot be interrupted. Therefore, it is advised to use this only for simple functions which do not take a long time.

Examples:

```
>>> from cypari2.closure import objtoclosure
>>> def pymul(i,j): return i*j
>>> mul = objtoclosure(pymul)
>>> mul
(v1,v2)->call_python(v1,v2,0,0,0,2,...)
>>> mul(6,9)
54
>>> mul.type()
't_CLOSURE'
>>> mul.arity()
```

```
2
>>> def printme(x):
... print(x)
>>> objtoclosure(printme)('matid(2)')
[1, 0; 0, 1]
```

Construct the Riemann zeta function using a closure:

```
>>> from cypari2 import Pari; pari = Pari()
>>> def coeffs(n):
... return [1 for i in range(n)]
>>> Z = pari.lfuncreate([coeffs, 0, [0], 1, 1, 1, 1])
>>> Z.lfun(2)
1.64493406684823
```

A trivial closure:

```
>>> f = pari(lambda x: x)
>>> f(10)
10
```

Test various kinds of errors:

```
>>> mul(4)
Traceback (most recent call last):
. . .
TypeError: pymul() ...
>>> mul(None, None)
Traceback (most recent call last):
. . .
ValueError: Cannot convert None to pari
>>> mul(*range(100))
Traceback (most recent call last):
. . .
PariError: call_python: too many parameters in user-defined function call
>>> mul([1], [2])
Traceback (most recent call last):
. . .
PariError: call_python: ...
```

CHAPTER

FIVE

HANDLING PARI ERRORS

AUTHORS:

- Peter Bruin (September 2013): initial version (Sage ticket #9640)
- Jeroen Demeyer (January 2015): use cb_pari_err_handle (Sage ticket #14894)

exception cypari2.handle_error.PariError

Error raised by PARI

errdata()

Return the error data (a t_ERROR gen) corresponding to this error.

EXAMPLES:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> try:
... pari('Mod(2,6)')**-1
... except PariError as e:
... E = e.errdata()
>>> E
error("impossible inverse in Fp_inv: Mod(2, 6).")
>>> E.component(2)
Mod(2, 6)
```

errnum()

Return the PARI error number corresponding to this exception.

EXAMPLES:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> try:
... pari('1/0')
... except PariError as err:
... print(err.errnum())
31
```

errtext()

Return the message output by PARI when this error occurred.

EXAMPLES:

```
>>> import cypari2
>>> pari = cypari2.Pari()
>>> try:
... pari('pi()')
... except PariError as e:
... print(e.errtext())
not a function in function call
```

SIX

CONVERT PARI OBJECTS TO/FROM PYTHON/C NATIVE TYPES

This modules contains the following conversion routines:

- integers, long integers <-> PARI integers
- list of integers -> PARI polynomials
- doubles -> PARI reals
- pairs of doubles -> PARI complex numbers

PARI integers are stored as an array of limbs of type pari_ulong (which are 32-bit or 64-bit integers). Depending on the kernel (GMP or native), this array is stored little-endian or big-endian. This is encapsulated in macros like int_W(): see section 4.5.1 of the PARI library manual.

Python integers of type int are just C longs. Python integers of type long are stored as a little-endian array of type digit with 15 or 30 bits used per digit. The internal format of a long is not documented, but there is some information in longintrepr.h.

Because of this difference in bit lengths, converting integers involves some bit shuffling.

cypari2.convert.gen_to_integer(x)

Convert a PARI gen to a Python int or long.

INPUT:

• x - a PARI t_INT, t_FRAC, t_REAL, a purely real t_COMPLEX, a t_INTMOD or t_PADIC (which are lifted).

Examples:

```
>>> from cypari2.convert import gen_to_integer
>>> from cypari2 import Pari
>>> pari = Pari()
>>> a = gen_to_integer(pari("12345")); a; type(a)
12345
<.... 'int'>
>>> gen_to_integer(pari("10^30")) == 10**30
True
>>> gen_to_integer(pari("19/5"))
3
>>> gen_to_integer(pari("1 + 0.0*I"))
1
>>> gen_to_integer(pari("3/2 + 0.0*I"))
1
>>> gen_to_integer(pari("Mod(-1, 11)"))
10
>>> gen_to_integer(pari("5 + 0(5^10)"))
```

```
5
>>> gen_to_integer(pari("Pol(42)"))
42
>>> gen_to_integer(pari("u"))
Traceback (most recent call last):
. . .
TypeError: unable to convert PARI object u of type t_POL to an integer
>>> s = pari("x + 0(x^2)")
>>> s
x + 0(x^{2})
>>> gen_to_integer(s)
Traceback (most recent call last):
. . .
TypeError: unable to convert PARI object x + O(x^2) of type t_SER to an integer
>>> gen_to_integer(pari("1 + I"))
Traceback (most recent call last):
. . .
TypeError: unable to convert PARI object 1 + I of type t_COMPLEX to an integer
```

Tests:

```
>>> gen_to_integer(pari("1.0 - 2^64")) == -18446744073709551615
True
>>> gen_to_integer(pari("1 - 2^64")) == -18446744073709551615
True
>>> import sys
>>> if sys.version_info.major == 3:
... long = int
>>> for i in range(10000):
... x = 3**i
... if long(pari(x)) != long(x) or int(pari(x)) != x:
... print(x)
```

Check some corner cases:

```
>>> for s in [1, -1]:
        for a in [1, 2**31, 2**32, 2**63, 2**64]:
. . .
             for b in [-1, 0, 1]:
. . .
                 Nstr = str(s * (a + b))
. . .
                 N1 = gen_to_integer(pari(Nstr)) # Convert via PARI
. . .
                 N2 = int(Nstr)
                                                     # Convert via Python
. . .
                 if N1 != N2:
. . .
                     print(Nstr, N1, N2)
. . .
                 if type(N1) is not type(N2):
. . .
                     print(N1, type(N1), N2, type(N2))
. . .
```

cypari2.convert.gen_to_python(z)

Convert the PARI element z to a Python object.

OUTPUT:

- a Python integer for integers (type t_INT)
- a Fraction (fractions module) for rationals (type t_FRAC)

- a float for real numbers (type t_REAL)
- a complex for complex numbers (type t_COMPLEX)
- a list for vectors (type t_VEC or t_COL). The function gen_to_python is then recursively applied on the entries.
- a list of Python integers for small vectors (type t_VECSMALL)
- a list of list``s for matrices (type ``t_MAT). The function gen_to_python is then recursively applied on the entries.
- the floating point inf or -inf for infinities (type t_INFINITY)
- a string for strings (type t_STR)
- other PARI types are not supported and the function will raise a NotImplementedError

Examples:

```
>>> from cypari2 import Pari
>>> from cypari2.convert import gen_to_python
>>> pari = Pari()
```

Converting integers:

```
>>> z = pari('42'); z
42
>>> a = gen_to_python(z); a
42
>>> type(a)
<... 'int'>
```

```
>>> gen_to_python(pari('3^50')) == 3**50
True
>>> type(gen_to_python(pari('3^50'))) == type(3**50)
True
```

Converting rational numbers:

```
>>> z = pari('2/3'); z
2/3
>>> a = gen_to_python(z); a
Fraction(2, 3)
>>> type(a)
<class 'fractions.Fraction'>
```

Converting real numbers (and infinities):

```
>>> z = pari('1.2'); z
1.2000000000000
>>> a = gen_to_python(z); a
1.2
>>> type(a)
<... 'float'>
```

```
>>> z = pari('oo'); z
+oo
>>> a = gen_to_python(z); a
inf
>>> type(a)
<... 'float'>
```

```
>>> z = pari('-oo'); z
-oo
>>> a = gen_to_python(z); a
-inf
>>> type(a)
<... 'float'>
```

Converting complex numbers:

```
>>> z = pari('1 + I'); z
1 + I
>>> a = gen_to_python(z); a
(1+1j)
>>> type(a)
<... 'complex'>
```

```
>>> z = pari('2.1 + 3.03*I'); z
2.1000000000000 + 3.0300000000000*I
>>> a = gen_to_python(z); a
(2.1+3.03j)
```

Converting vectors:

```
>>> z1 = pari('Vecsmall([1,2,3])'); z1
Vecsmall([1, 2, 3])
>>> z2 = pari('[1, 3.4, [-5, 2], oo]'); z2
[1, 3.400000000000, [-5, 2], +oo]
>>> z3 = pari('[1, 5.2]~'); z3
[1, 5.200000000000]~
>>> z1.type(), z2.type(), z3.type()
('t_VECSMALL', 't_VEC', 't_COL')
```

```
>>> a1 = gen_to_python(z1); a1
[1, 2, 3]
>>> type(a1)
<... 'list'>
>>> [type(x) for x in a1]
[<... 'int'>, <... 'int'>]
```

```
>>> a2 = gen_to_python(z2); a2
[1, 3.4, [-5, 2], inf]
>>> type(a2)
<... 'list'>
>>> [type(x) for x in a2]
[<... 'int'>, <... 'float'>, <... 'list'>, <... 'float'>]
```

>>> a3 = gen_to_python(z3); a3
[1, 5.2]
>>> type(a3)
<... 'list'>
>>> [type(x) for x in a3]
[<... 'int'>, <... 'float'>]

Converting matrices:

```
>>> z = pari('[1,2;3,4]')
>>> gen_to_python(z)
[[1, 2], [3, 4]]
```

```
>>> z = pari('[[1, 3], [[2]]; 3, [4, [5, 6]]]')
>>> gen_to_python(z)
[[[1, 3], [[2]]], [3, [4, [5, 6]]]]
```

Converting strings:

>>> z = pari('"Hello"')
>>> a = gen_to_python(z); a
'Hello'
>>> type(a)
<... 'str'>

Some currently unsupported types:

```
>>> z = pari('x')
>>> z.type()
't_POL'
>>> gen_to_python(z)
Traceback (most recent call last):
....
NotImplementedError: conversion not implemented for t_POL
```

```
>>> z = pari('12 + 0(2^13)')
>>> z.type()
't_PADIC'
>>> gen_to_python(z)
Traceback (most recent call last):
....
NotImplementedError: conversion not implemented for t_PADIC
```

cypari2.convert.integer_to_gen(x)

Convert a Python int or long to a PARI gen of type t_INT.

Examples:

```
>>> from cypari2.convert import integer_to_gen
>>> from cypari2 import Pari
>>> pari = Pari()
>>> a = integer_to_gen(int(12345)); a; type(a)
12345
```

```
<... 'cypari2.gen.Gen'>
>>> integer_to_gen(float(12345))
Traceback (most recent call last):
...
TypeError: integer_to_gen() needs an int or long argument, not float
>>> integer_to_gen(2**100)
1267650600228229401496703205376
```

Tests:

```
>>> import sys
>>> if sys.version_info.major == 3:
... long = int
>>> assert integer_to_gen(long(12345)) == 12345
>>> for i in range(10000):
... x = 3**i
... if pari(long(x)) != pari(x) or pari(int(x)) != pari(x):
... print(x)
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

С

cypari2.closure,769 cypari2.convert,774 cypari2.gen,382 cypari2.handle_error,772 cypari2.pari_instance,1 cypari2.stack,768

INDEX

А

abs() (cypari2.gen.Gen_base method), 430 abs() (cypari2.pari_instance.Pari_auto method), 19 acos() (cypari2.gen.Gen_base method), 430 acos() (cypari2.pari_instance.Pari_auto method), 19 acosh() (cypari2.gen.Gen base method), 430 acosh() (cypari2.pari_instance.Pari_auto method), 19 addhelp() (cypari2.pari_instance.Pari_auto method), 20 addprimes() (cypari2.gen.Gen_base method), 430 addprimes() (cypari2.pari_instance.Pari_auto method), 20 agm() (cypari2.gen.Gen_base method), 430 agm() (cypari2.pari_instance.Pari_auto method), 20 airy() (cypari2.gen.Gen_base method), 431 airy() (cypari2.pari_instance.Pari_auto method), 20 algadd() (cypari2.gen.Gen_base method), 431 algadd() (cypari2.pari_instance.Pari_auto method), 21 algalgtobasis() (cypari2.gen.Gen_base method), 431 algalgtobasis() (cypari2.pari_instance.Pari_auto method), 21 algaut() (cypari2.gen.Gen_base method), 431 algaut() (cypari2.pari instance.Pari auto method), 21 algb() (cypari2.gen.Gen_base method), 431 algb() (cypari2.pari instance.Pari auto method), 21 algbasis() (cypari2.gen.Gen_base method), 431 algbasis() (cypari2.pari_instance.Pari_auto method), 21 algbasistoalg() (cypari2.gen.Gen base method), 432 algbasistoalg() (cypari2.pari_instance.Pari_auto method), 22 algcenter() (cypari2.gen.Gen_base method), 432 (cypari2.pari_instance.Pari_auto algcenter() method), 22 algcentralproj() (cypari2.gen.Gen_base method), 432 algcentralproj() (cypari2.pari_instance.Pari_auto method), 22 algchar() (cypari2.gen.Gen_base method), 433 algchar() (cypari2.pari_instance.Pari_auto method), 23 algcharpoly() (cypari2.gen.Gen_base method), 433

algcharpoly() (cypari2.pari instance.Pari auto method), 23 algdegree() (cypari2.gen.Gen_base method), 433 algdegree() (cypari2.pari_instance.Pari_auto method), 23 algdep() (cypari2.gen.Gen_base method), 433 algdep() (cypari2.pari_instance.Pari_auto method), 23 algdim() (cypari2.gen.Gen_base method), 434 algdim() (cypari2.pari_instance.Pari_auto method), 24 algdisc() (cypari2.gen.Gen_base method), 434 algdisc() (cypari2.pari_instance.Pari_auto method), 24 algdivl() (cypari2.gen.Gen_base method), 435 algdivl() (cypari2.pari_instance.Pari_auto method), 25 algdivr() (cypari2.gen.Gen base method), 435 algdivr() (cypari2.pari_instance.Pari_auto method), 25 alggroup() (cypari2.gen.Gen_base method), 435 alggroup() (cypari2.pari_instance.Pari_auto method), 25 alggroupcenter() (cypari2.gen.Gen_base method), 435 alggroupcenter() (cypari2.pari_instance.Pari_auto method), 25 alghasse() (cypari2.gen.Gen_base method), 436 alghasse() (cypari2.pari_instance.Pari_auto method), 26 alghassef() (cypari2.gen.Gen_base method), 436 alghassef() (cypari2.pari_instance.Pari_auto method), 26 alghassei() (cypari2.gen.Gen_base method), 436 (cypari2.pari instance.Pari auto alghassei() method), 26 algindex() (cypari2.gen.Gen_base method), 436 algindex() (cypari2.pari_instance.Pari_auto method), 26alginit() (cypari2.gen.Gen_base method), 437 alginit() (cypari2.pari_instance.Pari_auto_method), 27 alginv() (cypari2.gen.Gen_base method), 439 alginv() (cypari2.pari_instance.Pari_auto method), 29

alginvbasis()(<i>cypar</i>	ri2.gen.Gen_base method), 439
alginvbasis()	(cypari2.pari_instance.Pari_auto
method), 29	
algisassociative()	(cypari2.gen.Gen_base method)

439 algisassociative() (cypari2.pari_instance.Pari_auto method). 29

- algiscommutative() (cypari2.gen.Gen_base method), 439
- algisdivision() (cypari2.gen.Gen_base method), 440
- algisdivision() (cypari2.pari_instance.Pari_auto method), 30
- algisdivl() (cypari2.gen.Gen_base method), 440
- algisdivl() (cypari2.pari_instance.Pari_auto method), 30
- algisinv() (cypari2.gen.Gen base method), 440
- algisinv() (cypari2.pari_instance.Pari_auto method), 30
- algisramified() (cypari2.gen.Gen_base method), 440
- algissemisimple() (*cypari2.gen.Gen_base method*), 441
- algissimple() (cypari2.gen.Gen_base method), 441
- algissimple() (cypari2.pari_instance.Pari_auto method), 31
- algissplit() (cypari2.gen.Gen_base method), 441
- alglatadd() (cypari2.gen.Gen_base method), 442
- alglatadd() (cypari2.pari_instance.Pari_auto method), 32
- alglatcontains() (cypari2.gen.Gen_base method), 442
- alglatelement() (cypari2.gen.Gen_base method), 442
 alglatelement() (cypari2.pari_instance.Pari_auto
- method), 32
- alglathnf() (cypari2.gen.Gen_base method), 443 alglathnf() (cypari2.pari_instance.Pari_auto
- method), 32
 alglatindex() (cypari2.gen.Gen_base method), 443
- alglatindex() (cypari2.gen.oen_base memora), ++5 alglatindex() (cypari2.pari_instance.Pari_auto
- method), 33
- alglatinter() (cypari2.gen.Gen_base method), 443
- alglatinter() (cypari2.pari_instance.Pari_auto method), 33
- alglatlefttransporter() (cypari2.gen.Gen_base method), 443

- alglatlefttransporter() (cypari2.pari_instance.Pari_auto method), 33
- alglatmul() (cypari2.gen.Gen_base method), 444
- alglatmul() (cypari2.pari_instance.Pari_auto method), 33
- alglatrighttransporter() (cypari2.pari_instance.Pari_auto method), 34
- alglatsubset() (cypari2.gen.Gen_base method), 444
- alglatsubset() (cypari2.pari_instance.Pari_auto method), 34

- algmul() (cypari2.gen.Gen_base method), 445
- algmul() (cypari2.pari_instance.Pari_auto method), 35
- algmultable() (cypari2.gen.Gen_base method), 445
- algmultable() (cypari2.pari_instance.Pari_auto method), 35
- algneg() (cypari2.gen.Gen_base method), 446
- algneg() (cypari2.pari_instance.Pari_auto method), 36
- algnorm() (cypari2.gen.Gen_base method), 446
- algnorm() (cypari2.pari_instance.Pari_auto method), 36
- algpoleval() (cypari2.gen.Gen_base method), 446
- algpoleval() (cypari2.pari_instance.Pari_auto method), 36
- algpow() (cypari2.gen.Gen_base method), 447
- algpow() (cypari2.pari_instance.Pari_auto method), 36
- algprimesubalg() (cypari2.gen.Gen_base method), 447
- algprimesubalg() (cypari2.pari_instance.Pari_auto method), 37
- algquotient() (cypari2.gen.Gen_base method), 447
- algradical() (cypari2.gen.Gen_base method), 447
- algradical() (cypari2.pari_instance.Pari_auto method), 37
- algramifiedplaces() (cypari2.gen.Gen_base method), 448
- algramifiedplaces() (cypari2.pari_instance.Pari_auto method), 38
- algrandom() (cypari2.gen.Gen_base method), 448
- algrandom() (cypari2.pari_instance.Pari_auto method), 38
- algrelmultable() (cypari2.gen.Gen_base method), 448
- algrelmultable() (cypari2.pari_instance.Pari_auto

method), 38 algsimpledec() (cypari2.gen.Gen base method), 449 algsimpledec() (cypari2.pari instance.Pari auto method), 38 algsplit() (cypari2.gen.Gen base method), 449 algsplit() (cypari2.pari instance.Pari auto method), 39 algsplittingdata() (cypari2.gen.Gen_base method), 450 algsplittingdata() (cypari2.pari_instance.Pari_auto method), 39 algsplittingfield() (cypari2.gen.Gen_base method), 450 algsplittingfield() (cypari2.pari_instance.Pari_auto method), 40 algsqr() (cypari2.gen.Gen_base method), 451 algsqr() (cypari2.pari instance.Pari auto method), 40 algsub() (cypari2.gen.Gen_base method), 451 algsub() (cypari2.pari instance.Pari auto method), 41 algsubalg() (cypari2.gen.Gen_base method), 451 algsubalg() (cypari2.pari instance.Pari auto method), 41 algtableinit() (cypari2.gen.Gen base method), 452 algtableinit() (cypari2.pari_instance.Pari_auto method), 41 algtensor() (cypari2.gen.Gen_base method), 452 algtensor() (cypari2.pari_instance.Pari_auto method), 42 algtomatrix() (cypari2.gen.Gen_base method), 452 algtomatrix() (cypari2.pari_instance.Pari_auto method), 42 algtrace() (cypari2.gen.Gen_base method), 453 algtrace() (cypari2.pari_instance.Pari_auto method), 43 algtype() (cypari2.gen.Gen base method), 453 algtype() (cypari2.pari instance.Pari auto method), 43 allocatemem() (cypari2.gen.Gen method), 386 allocatemem() (cypari2.pari_instance.Pari method), 5 apply() (cypari2.gen.Gen base method), 454 apply() (cypari2.pari instance.Pari auto method), 44 arg() (cypari2.gen.Gen base method), 455 arg() (cypari2.pari_instance.Pari_auto method), 45 arity() (cypari2.gen.Gen method), 387 arity() (cypari2.gen.Gen_base method), 455 arity() (cypari2.pari_instance.Pari_auto method), 45 asin() (cypari2.gen.Gen_base method), 455 asin() (cypari2.pari_instance.Pari_auto method), 45 asinh() (cypari2.gen.Gen_base method), 455 asinh() (cypari2.pari_instance.Pari_auto method), 45 asympnum() (cypari2.gen.Gen base method), 456 asympnum() (cypari2.pari_instance.Pari_auto method), 45

- asympnumraw() (cypari2.gen.Gen_base method), 457
- asympnumraw() (cypari2.pari_instance.Pari_auto method), 46
- atan() (cypari2.gen.Gen_base method), 457
- atan() (cypari2.pari_instance.Pari_auto method), 47
- atanh() (cypari2.gen.Gen_base method), 458
- atanh() (cypari2.pari_instance.Pari_auto method), 47

В

- 47
- bernvec() (cypari2.pari_instance.Pari_auto method), 48
- besselh1() (cypari2.gen.Gen_base method), 458
- besselh2() (cypari2.gen.Gen_base method), 458
- besseli() (cypari2.gen.Gen_base method), 458
- besselj() (cypari2.gen.Gen_base method), 458
- besseljh() (cypari2.gen.Gen_base method), 458
- besselk() (cypari2.gen.Gen method), 387
- besselk() (cypari2.gen.Gen_base method), 458
- besseln() (cypari2.gen.Gen_base method), 458
- bessely() (cypari2.gen.Gen_base method), 458
- bessely() (cypari2.pari_instance.Pari_auto method), 49
- bestappr() (cypari2.gen.Gen_base method), 458

- bezout() (cypari2.gen.Gen_base method), 460

bezout() (cypari2.pari instance.Pari auto method), 51 bnfisprincipal() (cypari2.gen.Gen_base method), bezoutres() (cypari2.gen.Gen_base method), 460 465 (cypari2.pari instance.Pari auto bnfisprincipal() (cypari2.pari instance.Pari auto bezoutres() method), 51 method), 55 bid_get_cyc() (cypari2.gen.Gen method), 388 bnfissunit() (cypari2.gen.Gen_base method), 466 bnfissunit() bid_get_gen() (cypari2.gen.Gen method), 388 (cypari2.pari instance.Pari auto bigomega() (cypari2.gen.Gen base method), 460 method), 56 bnfisunit() (cypari2.gen.Gen_base method), 466 bigomega() (cypari2.pari_instance.Pari_auto method), bnfisunit() 51 (cypari2.pari instance.Pari auto binary() (cypari2.gen.Gen_base method), 460 method), 56 binary() (cypari2.pari_instance.Pari_auto method), 51 bnflog() (cypari2.gen.Gen_base method), 467 binomial() (cypari2.gen.Gen_base method), 461 bnflog() (cypari2.pari_instance.Pari_auto method), 58 binomial() (cypari2.pari_instance.Pari_auto method), bnflogdegree() (cypari2.gen.Gen_base method), 468 51 bnflogdegree() (cypari2.pari_instance.Pari_auto bitand() (cypari2.gen.Gen_base method), 461 method), 58 bitand() (cypari2.pari_instance.Pari_auto method), 52 bnflogef() (cypari2.gen.Gen_base method), 468 bitneg() (cypari2.gen.Gen_base method), 461 bnflogef() (cypari2.pari_instance.Pari_auto method), bitneg() (cypari2.pari instance.Pari auto method), 52 58 bitnegimply() (cypari2.gen.Gen_base method), 461 bnfnarrow() (cypari2.gen.Gen_base method), 468 (cypari2.pari instance.Pari auto bitnegimply() bnfnarrow() (cypari2.pari instance.Pari auto method), 52 method), 59 bitor() (cypari2.gen.Gen_base method), 462 bnfsignunit() (cypari2.gen.Gen_base method), 468 bitor() (cypari2.pari_instance.Pari_auto method), 52 bnfsignunit() (cypari2.pari_instance.Pari_auto bitprecision() (cypari2.gen.Gen base method), 462 method). 59 bitprecision() (cypari2.pari instance.Pari auto bnfsunit() (cypari2.gen.Gen_base method), 469 method), 52 bnfsunit() (cypari2.pari_instance.Pari_auto method), bittest() (cypari2.gen.Gen method), 389 59 bittest() (cypari2.gen.Gen_base method), 462 bnfunit() (cypari2.gen.Gen method), 391 bittest() (cypari2.pari_instance.Pari_auto method), bnfunits() (cypari2.gen.Gen_base method), 469 53 bnfunits() (cypari2.pari_instance.Pari_auto method), bitxor() (cypari2.gen.Gen_base method), 462 bitxor() (cypari2.pari_instance.Pari_auto method), 53 bnrchar() (cypari2.gen.Gen_base method), 471 bnf_get_cyc() (cypari2.gen.Gen method), 389 bnrchar() (cypari2.pari_instance.Pari_auto method), bnf_get_fu() (cypari2.gen.Gen method), 389 61 bnf_get_gen() (cvpari2.gen.Gen method), 390 bnrclassfield() (cypari2.gen.Gen base method), 471 bnf_get_no() (cypari2.gen.Gen method), 390 bnrclassfield() (cypari2.pari_instance.Pari_auto bnf_get_reg() (cypari2.gen.Gen method), 390 method), 61 bnf_get_tu() (cypari2.gen.Gen method), 391 bnrclassno() (cypari2.gen.Gen_base method), 472 bnfcertify() (cypari2.gen.Gen_base method), 463 bnrclassno() (cypari2.pari_instance.Pari_auto bnfcertify() (cypari2.pari_instance.Pari_auto method), 62 bnrclassnolist() (cypari2.gen.Gen base method), method), 53 bnfdecodemodule() (cypari2.gen.Gen_base method), 472 bnrclassnolist() (cypari2.pari_instance.Pari_auto 463 bnfdecodemodule() (cypari2.pari_instance.Pari_auto method), 62 bnrconductor() (cypari2.gen.Gen_base method), 472 method), 54 bnfinit() (cypari2.gen.Gen_base method), 463 bnrconductor() (cypari2.pari_instance.Pari_auto bnfinit() (cypari2.pari_instance.Pari_auto method), method), 63 54 bnrconductorofchar() bnfisintnorm() (cypari2.gen.Gen_base method), 465 method), 473 (cypari2.pari_instance.Pari_auto bnfisintnorm() bnrconductorofchar() method), 55 pari2.pari_instance.Pari_auto bnfisnorm() (cypari2.gen.Gen_base method), 465 63 bnfisnorm() (cypari2.pari instance.Pari auto bnrdisc() (cypari2.gen.Gen_base method), 473 bnrdisc() (cypari2.pari instance.Pari auto method), method), 55

(cy-

method),

(cypari2.gen.Gen_base
63

- bnrdisclist() (cypari2.gen.Gen base method), 473
- bnrdisclist() (cypari2.pari_instance.Pari_auto method), 64

- bnrinit() (cypari2.gen.Gen_base method), 475

- bnrisgalois() (cypari2.gen.Gen_base method), 475
- bnrisgalois() (cypari2.pari_instance.Pari_auto method), 66

- bnrL1() (cypari2.gen.Gen_base method), 470
- bnrL1() (cypari2.pari_instance.Pari_auto method), 61
- bnrmap() (cypari2.gen.Gen_base method), 476
- bnrmap() (cypari2.pari_instance.Pari_auto method), 67
- bnrrootnumber() (cypari2.gen.Gen_base method), 477
- bnrstark() (cypari2.gen.Gen_base method), 477

С

- call() (cypari2.gen.Gen_base method), 478
- call() (cypari2.pari_instance.Pari_auto method), 68
- Catalan() (cypari2.pari_instance.Pari_auto method), 13
- ceil() (cypari2.gen.Gen_base method), 479
- ceil() (cypari2.pari_instance.Pari_auto method), 69
- centerlift() (cypari2.gen.Gen_base method), 479
- centerlift() (cypari2.pari_instance.Pari_auto
 method), 69

- characteristic() (cypari2.pari_instance.Pari_auto method), 69
- charconj() (cypari2.gen.Gen_base method), 479
- charconj() (cypari2.pari_instance.Pari_auto method), 69 chardiv() (cypari2.gen.Gen base method), 480 chardiv() (cypari2.pari_instance.Pari_auto method), 70 chareval() (cypari2.gen.Gen base method), 480 chareval() (cypari2.pari instance.Pari auto method), 71 chargalois() (cypari2.gen.Gen_base method), 481 chargalois() (cypari2.pari_instance.Pari_auto method), 71 charker() (cypari2.gen.Gen_base method), 481 charker() (cypari2.pari_instance.Pari_auto method), 72 charmul() (cypari2.gen.Gen_base method), 482 charmul() (cypari2.pari_instance.Pari_auto_method), 72 charorder() (cypari2.gen.Gen_base method), 482 charorder() (cypari2.pari_instance.Pari_auto method), 73 charpoly() (cypari2.gen.Gen_base method), 483 charpoly() (cypari2.pari_instance.Pari_auto method), 73 charpow() (cypari2.gen.Gen base method), 484 charpow() (cypari2.pari_instance.Pari_auto method), 74 chinese() (cypari2.gen.Gen_base method), 484 chinese() (cypari2.pari_instance.Pari_auto_method), 75 cmp() (cypari2.gen.Gen method), 392 cmp() (cypari2.gen.Gen_base method), 485 cmp() (cypari2.pari_instance.Pari_auto method), 75 Col() (cypari2.gen.Gen_base method), 424 Col() (cypari2.pari_instance.Pari_auto method), 13 Colrev() (cypari2.gen.Gen base method), 425 Colrev() (cypari2.pari_instance.Pari_auto method), 14 complex() (cypari2.pari instance.Pari method), 7 component() (cypari2.gen.Gen_base method), 485 component() (cypari2.pari_instance.Pari_auto method), 76 concat() (cypari2.gen.Gen base method), 486 concat() (cypari2.pari instance.Pari auto method), 76 conj() (cypari2.gen.Gen base method), 487 conj() (cypari2.pari_instance.Pari_auto method), 78 conjvec() (cypari2.gen.Gen_base method), 487 conjvec() (cypari2.pari_instance.Pari_auto method), 78 content() (cypari2.gen.Gen_base method), 487 content() (cypari2.pari_instance.Pari_auto method), 78
- contfrac() (cypari2.gen.Gen_base method), 488
- contfrac() (cypari2.pari_instance.Pari_auto method),
 79
- contfraceval() (cypari2.gen.Gen_base method), 489

contfraceval() (cypari2.pari_instance.Pari_auto method), 80 contfracinit() (cypari2.gen.Gen base method), 489 (cypari2.pari_instance.Pari_auto contfracinit() method), 80 contfracpnqn() (cypari2.gen.Gen base method), 489 contfracpnqn() (cypari2.pari instance.Pari auto method), 80 core() (cypari2.gen.Gen_base method), 490 core() (cypari2.pari_instance.Pari_auto method), 80 coredisc() (cypari2.gen.Gen_base method), 490 coredisc() (cypari2.pari_instance.Pari_auto method), 81 cos() (cypari2.gen.Gen_base method), 490 cos() (cypari2.pari_instance.Pari_auto method), 81 cosh() (cypari2.gen.Gen_base method), 490 cosh() (cypari2.pari_instance.Pari_auto method), 81 cotan() (cypari2.gen.Gen base method), 490 cotan() (cypari2.pari_instance.Pari_auto method), 81 cotanh() (cypari2.gen.Gen base method), 490 cotanh() (cypari2.pari_instance.Pari_auto method), 81 cypari2.closure module, 769 cypari2.convert module, 774 cypari2.gen module, 382 cypari2.handle_error module, 772 cypari2.pari_instance module, 1 cypari2.stack module, 768

D

debug() (cypari2.gen.Gen method), 393 debugstack() (cypari2.pari_instance.Pari method), 7 default() (cypari2.pari_instance.Pari_auto method), 81 default_bitprec() (in module cypari2.pari_instance), 380 denominator() (cypari2.gen.Gen_base method), 490 denominator() (cypari2.pari_instance.Pari_auto method), 81 deriv() (cypari2.gen.Gen base method), 491 deriv() (cypari2.pari_instance.Pari_auto method), 82 derivn() (cypari2.gen.Gen_base method), 492 derivn() (cypari2.pari_instance.Pari_auto method), 82 DetachGen (class in cypari2.stack), 769 diffop() (cypari2.gen.Gen base method), 492 diffop() (cypari2.pari instance.Pari auto method), 83 digits() (cypari2.gen.Gen base method), 493 digits() (cypari2.pari_instance.Pari_auto method), 83 dilog() (cypari2.gen.Gen_base method), 493

dirdiv() (cypari2.pari instance.Pari auto method), 84 dirmul() (cypari2.gen.Gen_base method), 493 dirmul() (cypari2.pari instance.Pari auto method), 84 dirpowers() (cypari2.pari instance.Pari auto method), 84 dirpowerssum() (cypari2.gen.Gen_base method), 493 dirpowerssum() (cypari2.pari instance.Pari auto method), 84 dirzetak() (cypari2.gen.Gen_base method), 494 dirzetak() (cypari2.pari_instance.Pari_auto method), 85 disc() (cypari2.gen.Gen method), 393 divisors() (cypari2.gen.Gen_base method), 494 divisors() (cypari2.pari_instance.Pari_auto method), 85 divisorslenstra() (cypari2.gen.Gen_base method), 495 divisorslenstra() (cypari2.pari instance.Pari auto method), 85 divrem() (cypari2.gen.Gen_base method), 495 divrem() (cypari2.pari_instance.Pari_auto method), 86 Е eint1() (cypari2.gen.Gen method), 393 eint1() (cypari2.gen.Gen base method), 495

dilog() (cvpari2.pari instance.Pari auto method), 84

dirdiv() (cypari2.gen.Gen_base method), 493

eint1() (cypari2.pari instance.Pari auto method), 86 elladd() (cypari2.gen.Gen_base method), 496 elladd() (cypari2.pari_instance.Pari_auto method), 87 ellak() (cypari2.gen.Gen_base method), 496 ellak() (cypari2.pari instance.Pari auto method), 87 ellan() (cypari2.gen.Gen method), 394 ellan() (cypari2.gen.Gen_base method), 497 ellan() (cypari2.pari_instance.Pari_auto method), 88 ellanalyticrank() (cypari2.gen.Gen_base method), 497 ellanalyticrank() (cypari2.pari_instance.Pari_auto method), 88 ellap() (cypari2.gen.Gen_base method), 497 ellap() (cypari2.pari_instance.Pari_auto method), 88 ellaplist() (cypari2.gen.Gen method), 394 ellbil() (cypari2.gen.Gen_base method), 499 ellbil() (cypari2.pari instance.Pari auto method), 90 ellbsd() (cypari2.gen.Gen base method), 499 ellbsd() (cypari2.pari_instance.Pari_auto method), 90 ellcard() (cypari2.gen.Gen_base method), 499 ellcard() (cypari2.pari_instance.Pari_auto method), 90

ellchangecurve() (cypari2.gen.Gen_base method), 500

ellchangecurve() (cypari2.pari_instance.Pari_auto method), 91 ellchangepoint() (cypari2.gen.Gen_base method), ellgroup() (cypari2.pari_instance.Pari_auto method), 500 ellchangepoint() (cypari2.pari instance.Pari auto method), 91 ellchangepointinv() (cypari2.gen.Gen_base method), 500 ellchangepointinv() (cypari2.pari_instance.Pari_auto *method*), 91 ellconvertname() (cypari2.gen.Gen_base method), 500 ellconvertname() (cypari2.pari_instance.Pari_auto method), 91 elldivpol() (cypari2.gen.Gen_base method), 500 elldivpol() (cypari2.pari_instance.Pari_auto method), 91 ellE() (cypari2.gen.Gen_base method), 495 ellE() (cypari2.pari instance.Pari auto method), 86 elleisnum() (cypari2.gen.Gen_base method), 501 (cypari2.pari instance.Pari auto elleisnum() method), 92 elleta() (cypari2.gen.Gen base method), 501 elleta() (cypari2.pari_instance.Pari_auto method), 92 ellformaldifferential() (cypari2.gen.Gen base method), 501 ellformaldifferential() (cypari2.pari_instance.Pari_auto *method*), 92 ellformalexp() (cypari2.gen.Gen_base method), 502 ellformalexp() (cypari2.pari_instance.Pari_auto method), 93 ellformallog() (cypari2.gen.Gen_base method), 502 (cypari2.pari_instance.Pari_auto ellformallog() method), 93 ellformalpoint() (cypari2.gen.Gen_base method), 502 ellformalpoint() (cypari2.pari instance.Pari auto method), 93 ellformalw() (cypari2.gen.Gen_base method), 502 ellformalw() (cypari2.pari_instance.Pari_auto method), 93 ellfromeqn() (cypari2.gen.Gen base method), 503 ellfromean() (cypari2.pari_instance.Pari_auto method), 94 ellfromj() (cypari2.gen.Gen_base method), 503 ellfromj() (cypari2.pari_instance.Pari_auto method), 94 ellgenerators() (cypari2.gen.Gen_base method), 503 ellgenerators() (cypari2.pari_instance.Pari_auto method), 94 ellglobalred() (cypari2.gen.Gen_base method), 503 ellglobalred() (cypari2.pari_instance.Pari_auto method), 94 ellgroup() (cypari2.gen.Gen_base method), 504

95 ellheegner() (cypari2.gen.Gen base method), 505 ellheegner() (cypari2.pari_instance.Pari_auto method), 96 ellheight() (cypari2.gen.Gen base method), 505 ellheight() (cypari2.pari instance.Pari auto method), 96 ellheightmatrix() (cypari2.gen.Gen_base method), 505 ellheightmatrix() (cypari2.pari_instance.Pari_auto method), 96 ellidentify() (cypari2.gen.Gen_base method), 506 ellidentify() (cypari2.pari_instance.Pari_auto method), 96 ellinit() (cypari2.gen.Gen_base method), 506 ellinit() (cypari2.pari_instance.Pari_auto method), 97 ellintegralmodel() (cypari2.gen.Gen_base method), 507 ellintegralmodel() (cypari2.pari_instance.Pari_auto method), 98 ellisdivisible() (cypari2.gen.Gen_base method), 507 ellisdivisible() (cypari2.pari_instance.Pari_auto method), 98 ellisogeny() (cypari2.gen.Gen_base method), 508 (cypari2.pari_instance.Pari_auto ellisogeny() method), 99 ellisogenyapply() (cypari2.gen.Gen_base method), 508 ellisogenyapply() (cypari2.pari_instance.Pari_auto method), 99 ellisomat() (cypari2.gen.Gen_base method), 508 ellisomat() (cypari2.pari_instance.Pari_auto method), 99 ellisoncurve() (cypari2.gen.Gen method), 395 ellisoncurve() (cypari2.gen.Gen_base method), 509 (cypari2.pari_instance.Pari_auto ellisoncurve() method), 100 ellisotree() (cypari2.gen.Gen base method), 509 ellisotree() (cypari2.pari_instance.Pari_auto method), 100 ellissupersingular() (cypari2.gen.Gen_base method), 510 ellissupersingular() (cypari2.pari_instance.Pari_auto *method*). 101 ellj() (cypari2.gen.Gen_base method), 511 ellj() (cypari2.pari_instance.Pari_auto method), 102 ellK() (cypari2.gen.Gen_base method), 496

- ellL1() (cypari2.gen.Gen base method), 496
- ellL1() (cypari2.pari instance.Pari auto method), 87

elllocalred() (cypari2.gen.Gen_base method), 511 elllocalred() (cypari2.pari instance.Pari auto method), 102 elllog() (cypari2.gen.Gen_base method), 511 elllog() (cypari2.pari_instance.Pari_auto method), 102 elllseries() (cypari2.gen.Gen base method), 511 (cypari2.pari_instance.Pari_auto elllseries() method), 102 ellminimaldisc() (cypari2.gen.Gen_base method), 512 ellminimaldisc() (cypari2.pari_instance.Pari_auto method), 102 ellminimalmodel() (cypari2.gen.Gen method), 395 ellminimalmodel() (cypari2.gen.Gen_base method), 512 ellminimalmodel() (cypari2.pari_instance.Pari_auto method), 103 ellminimaltwist() (cypari2.gen.Gen_base method), 512 ellminimaltwist() (cypari2.pari_instance.Pari_auto method), 103 ellmoddegree() (cypari2.gen.Gen_base method), 513 (cypari2.pari_instance.Pari_auto ellmoddegree() method), 104 ellmodulareqn() (cypari2.pari_instance.Pari_auto method), 104 ellmul() (cypari2.gen.Gen_base method), 513 ellmul() (cypari2.pari_instance.Pari_auto method), 105 ellneg() (cypari2.gen.Gen_base method), 513 ellneg() (cypari2.pari_instance.Pari_auto method), 105 ellnonsingularmultiple() (cypari2.gen.Gen_base method), 514 ellnonsingularmultiple() (cypari2.pari instance.Pari auto *method*), 105 ellorder() (cypari2.gen.Gen_base method), 514 ellorder() (cypari2.pari_instance.Pari_auto method), 105 ellordinate() (cypari2.gen.Gen_base method), 514 (cypari2.pari_instance.Pari_auto ellordinate() method), 106 ellpadicbsd() (cypari2.gen.Gen_base method), 516 (cypari2.pari_instance.Pari_auto ellpadicbsd() method), 108 ellpadicfrobenius() (cypari2.gen.Gen_base method), 517 ellpadicfrobenius() (cypari2.pari_instance.Pari_auto *method*), 109 ellpadicheight() (cypari2.gen.Gen_base method), elltamagawa() (cypari2.gen.Gen_base method), 525 518

ellpadicheight() (cypari2.pari_instance.Pari_auto method). 109 ellpadicheightmatrix() (cypari2.gen.Gen base method), 519 ellpadicheightmatrix() (cypari2.pari instance.Pari auto *method*), 111 ellpadicL() (cypari2.gen.Gen_base method), 514 ellpadicL() (cypari2.pari instance.Pari auto method), 106 ellpadiclambdamu() (cypari2.gen.Gen_base method), 519 ellpadiclambdamu() (cypari2.pari_instance.Pari_auto *method*), 111 ellpadiclog() (cypari2.gen.Gen_base method), 520 ellpadiclog() (cypari2.pari_instance.Pari_auto method), 112 ellpadicregulator() (cypari2.gen.Gen_base method), 520 ellpadicregulator() (cypari2.pari_instance.Pari_auto method), 112 ellpadics2() (cypari2.gen.Gen_base method), 521 ellpadics2() (cypari2.pari instance.Pari auto method), 112 ellperiods() (cypari2.gen.Gen_base method), 521 ellperiods() (cypari2.pari_instance.Pari_auto method), 113 ellpointtoz() (cypari2.gen.Gen_base method), 522 ellpointtoz() (cypari2.pari_instance.Pari_auto method), 113 ellpow() (cypari2.gen.Gen_base method), 523 ellpow() (cypari2.pari_instance.Pari_auto method), 114 ellratpoints() (cypari2.gen.Gen base method), 523 ellratpoints() (cypari2.pari_instance.Pari_auto method), 114 ellrootno() (cypari2.gen.Gen_base method), 523 ellrootno() (cypari2.pari_instance.Pari_auto method), 114 ellsea() (cypari2.gen.Gen base method), 523 ellsea() (cypari2.pari instance.Pari auto method), 115 ellsearch() (cypari2.gen.Gen_base method), 524 ellsearch() (cypari2.pari_instance.Pari_auto method), 116 ellsigma() (cypari2.gen.Gen_base method), 525 ellsigma() (cypari2.pari_instance.Pari_auto method), 116 ellsub() (cypari2.gen.Gen_base method), 525 ellsub() (cypari2.pari_instance.Pari_auto method), 116

elltamagawa() (cypari2.pari instance.Pari auto method), 116

- elltaniyama() (cypari2.gen.Gen_base method), 525
- elltaniyama() (cypari2.pari_instance.Pari_auto method), 117

- elltors() (cypari2.gen.Gen method), 396
- elltors() (cypari2.gen.Gen_base method), 525
- elltors() (cypari2.pari_instance.Pari_auto method), 117
- elltwist() (cypari2.gen.Gen_base method), 526
- ellweilcurve() (cypari2.gen.Gen_base method), 526
- ellweilcurve() (cypari2.pari_instance.Pari_auto method), 118

- ellwp() (cypari2.gen.Gen method), 396
- ellwp() (cypari2.gen.Gen_base method), 527
- ellwp() (cypari2.pari_instance.Pari_auto method), 118
- ellxn() (cypari2.gen.Gen_base method), 528
- ellxn() (cypari2.pari_instance.Pari_auto method), 119
- ellzeta() (cypari2.gen.Gen_base method), 528
- ellzeta() (cypari2.pari_instance.Pari_auto method), 119
- ellztopoint() (cypari2.gen.Gen_base method), 528
- erfc() (cypari2.gen.Gen_base method), 529
- erfc() (cypari2.pari_instance.Pari_auto method), 121
- errdata() (cypari2.handle_error.PariError method),
 773
- errname() (cypari2.gen.Gen_base method), 529
- errname() (cypari2.pari_instance.Pari_auto method), 121
- errnum() (cypari2.handle_error.PariError method), 773
- errtext() (cypari2.handle_error.PariError method),
 773
- eta() (cypari2.gen.Gen_base method), 530
- eta() (cypari2.pari_instance.Pari_auto method), 121
- euler() (cypari2.pari_instance.Pari method), 7
- Euler() (cypari2.pari_instance.Pari_auto method), 14

- eulerphi() (cypari2.gen.Gen_base method), 530

- eval() (cypari2.gen.Gen method), 397
- exp() (cypari2.gen.Gen_base method), 530
- exp() (cypari2.pari_instance.Pari_auto method), 122
- expm1() (cypari2.gen.Gen_base method), 530
- expm1() (cypari2.pari_instance.Pari_auto method), 122
- exponent() (cypari2.gen.Gen_base method), 530

F

- factor() (cypari2.gen.Gen method), 400
- factor() (cypari2.gen.Gen_base method), 531
- factorback() (cypari2.gen.Gen_base method), 536
- factorback() (cypari2.pari_instance.Pari_auto
 method), 129
- factorcantor() (cypari2.gen.Gen_base method), 536
- factorcantor() (cypari2.pari_instance.Pari_auto
 method), 129
- factorff() (cypari2.gen.Gen_base method), 536
- factorial() (cypari2.pari_instance.Pari_auto
 method), 129
- factorial_int() (cypari2.pari_instance.Pari method),
 7
- factorint() (cypari2.gen.Gen_base method), 537
- factorint() (cypari2.pari_instance.Pari_auto
 method), 129
- factormod() (cypari2.gen.Gen_base method), 537
- factormodDDF() (cypari2.gen.Gen_base method), 538
- factormodSQF() (cypari2.gen.Gen_base method), 539
- factornf() (cypari2.gen.Gen_base method), 540
- factornf() (cypari2.pari_instance.Pari_auto method),
 132
- factorpadic() (cypari2.gen.Gen method), 401
- factorpadic() (cypari2.gen.Gen_base method), 540

- factorpadic() (cypari2.pari instance.Pari auto method), 133 ffcompomap() (cypari2.gen.Gen base method), 541 ffcompomap() (cypari2.pari_instance.Pari_auto method), 133 ffembed() (cypari2.gen.Gen base method), 541 ffembed() (cypari2.pari instance.Pari auto method), 133 ffextend() (cypari2.gen.Gen base method), 541 ffextend() (cypari2.pari_instance.Pari_auto method), 134 fffrobenius() (cypari2.gen.Gen_base method), 541 (cypari2.pari_instance.Pari_auto fffrobenius() method), 134 ffgen() (cypari2.gen.Gen_base method), 542 ffgen() (cypari2.pari_instance.Pari_auto method), 134 ffinit() (cypari2.gen.Gen_base method), 542 ffinit() (cypari2.pari_instance.Pari_auto method), 135 ffinvmap() (cypari2.gen.Gen base method), 542 ffinvmap() (cypari2.pari_instance.Pari_auto method), 135 fflog() (cypari2.gen.Gen_base method), 543 fflog() (cypari2.pari instance.Pari auto method), 135 ffmap() (cypari2.gen.Gen base method), 543 ffmap() (cypari2.pari instance.Pari auto method), 136 ffmaprel() (cypari2.gen.Gen_base method), 543 ffmaprel() (cypari2.pari_instance.Pari_auto method), 136 ffnbirred() (cypari2.gen.Gen_base method), 544 ffnbirred() (cypari2.pari_instance.Pari_auto method), 136 fforder() (cypari2.gen.Gen_base method), 544 fforder() (cypari2.pari_instance.Pari_auto method), 136 ffprimroot() (cypari2.gen.Gen method), 402 ffprimroot() (cypari2.gen.Gen_base method), 544 ffprimroot() (cypari2.pari_instance.Pari_auto method), 137 fft() (cypari2.gen.Gen_base method), 545 fft() (cypari2.pari instance.Pari auto method), 137 fftinv() (cypari2.gen.Gen base method), 545 fftinv() (cypari2.pari_instance.Pari_auto method), 138 fibonacci() (cypari2.gen.Gen method), 402 fibonacci() (cypari2.pari_instance.Pari_auto method), 138 fileclose() (cypari2.pari_instance.Pari_auto method), 138 fileextern() (cypari2.pari_instance.Pari_auto method), 139 fileflush() (cypari2.gen.Gen_base method), 546 fileflush() (cypari2.pari instance.Pari auto method), 139
 - fileopen() (cypari2.pari_instance.Pari_auto method), 139
 - fileread() (cypari2.pari instance.Pari auto method), 140
 - filereadstr() (cypari2.pari_instance.Pari_auto method), 140
 - filewrite() (cypari2.pari instance.Pari auto method), 140
 - filewrite1() (cypari2.pari_instance.Pari_auto method), 141
 - floor() (cypari2.gen.Gen_base method), 546
 - floor() (cypari2.pari_instance.Pari_auto method), 141
 - fold() (cypari2.gen.Gen_base method), 546
 - fold() (cypari2.pari_instance.Pari_auto method), 141
 - frac() (cypari2.gen.Gen_base method), 546
 - frac() (cypari2.pari_instance.Pari_auto method), 142
 - fromdigits() (cypari2.gen.Gen_base method), 546
 - fromdigits() (cypari2.pari_instance.Pari_auto method), 142

G

galoischardet() (cypari2.gen.Gen base method), 547

- galoischardet() (cypari2.pari_instance.Pari_auto method), 142
- galoischarpoly() (cypari2.gen.Gen_base method), 547
- galoischarpoly() (cypari2.pari_instance.Pari_auto method), 142
- galoischartable() (cypari2.gen.Gen_base method), 547
- galoischartable() (cypari2.pari_instance.Pari_auto method), 143
- galoisconjclasses() (cypari2.gen.Gen_base method), 548
- galoisconjclasses() (cypari2.pari_instance.Pari_auto *method*), 144
- galoisexport() (cypari2.gen.Gen_base method), 549

galoisexport() (cypari2.pari instance.Pari auto method), 144

- galoisfixedfield() (cypari2.gen.Gen base method), 549
- galoisfixedfield() (cypari2.pari_instance.Pari_auto method), 145
- galoisgetgroup() (cypari2.pari_instance.Pari_auto method), 145
- galoisgetname() (cypari2.pari_instance.Pari_auto method), 145
- galoisgetpol() (cypari2.pari_instance.Pari_auto method), 146
- galoisidentify() (cypari2.gen.Gen_base method), 550
- galoisidentify() (cypari2.pari_instance.Pari_auto method), 146

- galoisinit() (cypari2.gen.Gen_base method), 550

- galoisisnormal() (cypari2.gen.Gen_base method),
 551
- galoispermtopol() (cypari2.gen.Gen_base method),
 551
- galoissubcyclo() (cypari2.gen.Gen_base method),
 551
- galoissubfields() (cypari2.gen.Gen method), 402

- galoissubgroups() (cypari2.gen.Gen_base method),
 552
- gamma() (cypari2.gen.Gen_base method), 552
- gamma() (cypari2.pari_instance.Pari_auto method), 149
- gammah() (cypari2.gen.Gen_base method), 553
- gammah() (cypari2.pari_instance.Pari_auto method), 149
- gammamellininv() (cypari2.gen.Gen_base method),
 553
- gammamellininvasymp() (cypari2.gen.Gen_base method), 553
- gammamellininvasymp() (cypari2.pari_instance.Pari_auto method), 149
- gammamellininvinit() (cypari2.pari_instance.Pari_auto method), 149
- gcd() (cypari2.gen.Gen_base method), 554
- gcd() (cypari2.pari_instance.Pari_auto method), 150
- gcdext() (cypari2.gen.Gen_base method), 554
- Gen (*class in cypari2.gen*), 383
- Gen_base (class in cypari2.gen), 424

- gen_to_integer() (in module cypari2.convert), 775
- gen_to_python() (in module cypari2.convert), 776
- genus2red() (cypari2.gen.Gen_base method), 555
- genus2red() (cypari2.pari_instance.Pari method), 7
- genus2red() (cypari2.pari_instance.Pari_auto method), 151
- gequal() (cypari2.gen.Gen method), 403
- gequal0() (*cypari2.gen.Gen method*), 403
- gequal_long() (cypari2.gen.Gen method), 404

- get_real_precision_bits() (cypari2.pari_instance.Pari method), 8
- getabstime() (cypari2.pari_instance.Pari_auto method), 153
- getattr() (cypari2.gen.Gen method), 404
- getcache() (cypari2.pari_instance.Pari_auto method),
 153
- getenv() (cypari2.pari_instance.Pari_auto method),
 153

- getwalltime() (cypari2.pari_instance.Pari_auto method), 154

Н

- halfgcd() (cypari2.gen.Gen base method), 556
- halfgcd() (cypari2.pari_instance.Pari_auto method), 154
- hilbert() (cypari2.gen.Gen_base method), 557
- hilbert() (cypari2.pari_instance.Pari_auto method),
 155
- hyperellcharpoly() (cypari2.gen.Gen_base method), 557

- hyperellpadicfrobenius() (cypari2.pari_instance.Pari_auto method), 155
- hyperellratpoints() (cypari2.gen.Gen_base method), 557
- hyperellratpoints() (cypari2.pari_instance.Pari_auto method), 155
- hypergeom() (cypari2.gen.Gen_base method), 558
- hypergeom() (cypari2.pari_instance.Pari_auto method), 156
- hyperu() (cypari2.gen.Gen_base method), 559
- hyperu() (cypari2.pari_instance.Pari_auto method), 157

I

I() (cypari2.pari_instance.Pari_auto method), 14 idealadd() (cypari2.gen.Gen_base method), 559 idealadd() (cypari2.pari_instance.Pari_auto method), 157 idealaddtoone() (cypari2.gen.Gen_base method), 560 idealaddtoone() (cypari2.pari_instance.Pari_auto method), 158 idealappr() (cypari2.gen.Gen_base method), 560 idealappr() (cypari2.pari instance.Pari auto method), 158 idealchinese() (cypari2.gen.Gen_base method), 560 idealchinese() (cypari2.pari instance.Pari auto method), 158 idealcoprime() (cypari2.gen.Gen_base method), 561 (cypari2.pari_instance.Pari_auto idealcoprime() method), 159 idealdiv() (cypari2.gen.Gen_base method), 561 idealdiv() (cypari2.pari_instance.Pari_auto method), 159 idealdown() (cypari2.gen.Gen_base method), 561 idealdown() (cypari2.pari_instance.Pari_auto method), 159 idealfactor() (cypari2.gen.Gen_base method), 561 idealfactor() (cypari2.pari instance.Pari auto method), 159 idealfactorback() (cypari2.gen.Gen_base method), 562 idealfactorback() (cypari2.pari_instance.Pari_auto method), 160 idealfrobenius() (cypari2.gen.Gen_base method), 563 idealfrobenius() (cypari2.pari_instance.Pari_auto *method*), 161 idealhnf() (cypari2.gen.Gen_base method), 563 idealhnf() (cypari2.pari_instance.Pari_auto method), 161 idealintersect() (cypari2.gen.Gen_base method), 564

idealintersect() (cypari2.pari_instance.Pari_auto method), 162 idealinv() (cypari2.gen.Gen base method), 565 idealinv() (cypari2.pari_instance.Pari_auto method), 162 idealismaximal() (cypari2.gen.Gen base method), 565 idealismaximal() (cypari2.pari_instance.Pari_auto method), 162 idealispower() (cypari2.gen.Gen_base method), 565 idealispower() (cypari2.pari_instance.Pari_auto method), 163 ideallist() (cypari2.gen.Gen_base method), 565 ideallist() (cypari2.pari_instance.Pari_auto method), 163 ideallistarch() (cypari2.gen.Gen_base method), 566 ideallistarch() (cypari2.pari_instance.Pari_auto method), 164 ideallog() (cypari2.gen.Gen_base method), 567 ideallog() (cypari2.pari instance.Pari auto method), 165 idealmin() (cypari2.gen.Gen_base method), 567 idealmin() (cypari2.pari_instance.Pari_auto method), 165 idealmoddivisor() (cypari2.gen.Gen method), 404 idealmul() (cypari2.gen.Gen base method), 567 idealmul() (cypari2.pari_instance.Pari_auto method), 165 idealnorm() (cypari2.gen.Gen_base method), 567 idealnorm() (cypari2.pari_instance.Pari_auto method), 165 idealnumden() (cypari2.gen.Gen_base method), 567 idealnumden() (cypari2.pari_instance.Pari_auto method), 165 idealpow() (cypari2.gen.Gen base method), 568 idealpow() (cypari2.pari_instance.Pari_auto method), 165 idealprimedec() (cypari2.gen.Gen_base method), 568 idealprimedec() (cypari2.pari_instance.Pari_auto method), 166 idealprincipalunits() (cypari2.gen.Gen base method), 568 idealprincipalunits() (cypari2.pari_instance.Pari_auto method), 166 idealramgroups() (cypari2.gen.Gen_base method), 569 idealramgroups() (cypari2.pari_instance.Pari_auto method), 166 idealred() (cypari2.gen.Gen_base method), 569 idealred() (cypari2.pari_instance.Pari_auto method), 167

idealredmodpower() (cypari2.gen.Gen_base method),
570

idealredmodpower() (cypari2.pari instance.Pari auto ispseudoprime() (cypari2.gen.Gen base method), 575 method), 168 idealstar() (cypari2.gen.Gen base method), 570 (cypari2.pari_instance.Pari_auto idealstar() method), 168 idealtwoelt() (cypari2.gen.Gen base method), 571 idealtwoelt() (cypari2.pari instance.Pari auto method), 169 idealval() (cypari2.gen.Gen base method), 572 idealval() (cypari2.pari_instance.Pari_auto method), 169 imag() (cypari2.gen.Gen_base method), 572 imag() (cypari2.pari_instance.Pari_auto method), 169 incgam() (cypari2.gen.Gen_base method), 572 incgam() (cypari2.pari_instance.Pari_auto method), 170 incgamc() (cypari2.gen.Gen_base method), 572 incgamc() (cypari2.pari_instance.Pari_auto method), 170 init_primes() (cypari2.pari instance.Pari method), 8 input() (cypari2.pari_instance.Pari_auto method), 170 install() (cypari2.pari_instance.Pari_auto method), 170 integer_to_gen() (in module cypari2.convert), 779 intformal() (cypari2.gen.Gen_base method), 572 (cypari2.pari_instance.Pari_auto intformal() method), 171 intnumgaussinit() (cypari2.pari_instance.Pari_auto method), 171 intnuminit() (cypari2.gen.Gen_base method), 573 intnuminit() (cypari2.pari_instance.Pari_auto method), 172 isfundamental() (cypari2.gen.Gen_base method), 573 isfundamental() (cypari2.pari_instance.Pari_auto method), 173 ispolygonal() (cypari2.gen.Gen_base method), 573 ispolygonal() (cypari2.pari instance.Pari auto method), 173 ispower() (cypari2.gen.Gen method), 405 ispower() (cypari2.gen.Gen_base method), 574 ispower() (cypari2.pari instance.Pari auto method), 173 ispowerful() (cypari2.gen.Gen base method), 574 ispowerful() (cypari2.pari_instance.Pari_auto method), 173 isprime() (cypari2.gen.Gen method), 405 isprime() (cypari2.gen.Gen_base method), 574 isprime() (cypari2.pari_instance.Pari_auto method), 174 isprimepower() (cypari2.gen.Gen method), 406 isprimepower() (cypari2.gen.Gen_base method), 574 (cypari2.pari_instance.Pari_auto isprimepower() method), 174 ispseudoprime() (cypari2.gen.Gen method), 406

<pre>ispseudoprime() (cypari2.pari_instance.Pari_auto</pre>		
<pre>ispseudoprimepower() (cypari2.gen.Gen method),</pre>		
<pre>ispseudoprimepower() (cypari2.gen.Gen_base</pre>		
<pre>ispseudoprimepower() (cy-</pre>		
pari2.pari_instance.Pari_auto method), 175		
<pre>issquare() (cypari2.gen.Gen method), 407</pre>		
<pre>issquare() (cypari2.gen.Gen_base method), 575</pre>		
<pre>issquare() (cypari2.pari_instance.Pari_auto method),</pre>		
175		
<pre>issquarefree() (cypari2.gen.Gen method), 407</pre>		
<pre>issquarefree() (cypari2.gen.Gen_base method), 576</pre>		
<pre>issquarefree() (cypari2.pari_instance.Pari_auto</pre>		
<pre>istotient() (cypari2.gen.Gen_base method), 576</pre>		
<pre>istotient() (cypari2.pari_instance.Pari_auto</pre>		
method), 176		

J

j() (cypari2.gen.Gen method), 408

Κ

kill() (cypari2.pari_instance.Pari_auto method), 176 kronecker() (cypari2.gen.Gen_base method), 576 kronecker() (cypari2.pari_instance.Pari_auto method), 177

L

- lambertw() (cypari2.gen.Gen base method), 577 lambertw() (cypari2.pari_instance.Pari_auto method), 177 laurentseries() (cypari2.gen.Gen_base method), 577 (cypari2.pari_instance.Pari_auto laurentseries() method), 177 lcm() (cypari2.gen.Gen_base method), 577 lcm() (cypari2.pari_instance.Pari_auto method), 178 length() (cypari2.gen.Gen_base method), 578 length() (cypari2.pari_instance.Pari_auto method), 178 lex() (cypari2.gen.Gen_base method), 578 lex() (cypari2.pari instance.Pari auto method), 179 lfun() (cypari2.gen.Gen_base method), 579 lfun() (cypari2.pari instance.Pari auto method), 179 lfunabelianrelinit() (cypari2.gen.Gen_base method), 579 lfunabelianrelinit() (cypari2.pari_instance.Pari_auto method), 180
- lfunan() (cypari2.gen.Gen_base method), 580

lfunan() (cypari2.pari_instance.Pari_auto method), 180	lfunroo
lfunartin() (cypari2.gen.Gen_base method), 580lfunartin() (cypari2.pari_instance.Pari_auto	lfunshi lfunshi
method), 180 1funcheckfeq() (cypari2.gen.Gen_base method), 581 1funcheckfeq() (cypari2 pari instance Pari auto	lfunsym
method), 181	TTunsyn
<pre>lfunconductor() (cypari2.gen.Gen_base method), 581 lfunconductor() (cypari2.pari_instance.Pari_auto</pre>	lfunthe lfunthe
lfuncost() (cypari2.gen.Gen_base method), 583 lfuncost() (cypari2.pari_instance.Pari_auto method), 183	lfunthe lfunthe
<pre>lfuncreate() (cypari2.gen.Gen_base method), 584 lfuncreate() (cypari2.pari_instance_Pari_auto</pre>	lfunthe lfunthe
method), 184	
<pre>lfundiv() (cypari2.gen.Gen_base method), 585 lfundiv() (cypari2.pari_instance.Pari_auto method),</pre>	lfuntwi lfuntwi
<pre>lfundual() (cypari2.gen.Gen_base method), 586 lfundual() (cypari2.pari_instance.Pari_auto method), 186</pre>	lfunzer lfunzer
lfunetaquo() (cypari2.gen.Gen_base method), 586	lift()
lfunetaquo() (cypari2.pari_instance.Pari_auto	lift()(
method), 180	lift_ce
11ungenus2() (cypari2.gen.Gen_base method), 580 1fungenus2() (cypari2.pari_instance.Pari_auto method) 187	liftall
1 funhardy() (cypari2.gen.Gen_base_method), 586	liftint
lfunhardy() (cypari2.pari_instance.Pari_auto method), 187	liftint
lfuninit() (cypari2.gen.Gen base method), 587	liftpol
lfuninit() (cypari2.pari_instance.Pari_auto method), 187	liftpol
lfunlambda() (cypari2.gen.Gen_base method), 587	limitnu
lfunlambda() (cypari2.pari_instance.Pari_auto method), 187	limitnu
lfunmf() (cypari2.gen.Gen_base method), 587	lindep(
lfunmf() (cypari2.pari_instance.Pari_auto method), 188	lindep(
<pre>lfunmfspec() (cypari2.gen.Gen_base method), 587</pre>	list()(
lfunmfspec() (cypari2.pari_instance.Pari_auto	List()(
<i>method</i>), 188	List()(
funmul() (cypari2.gen.Gen_base method), 588	L1St()(
11 [1] 11 [1] 12	listins
Ifunorderzero() (cypari2.gen.Gen_base method), 588	
<i>method</i>), 188	listkil listkil
fungt() (cypari2.gen.Gen_base method), 588	1:0+00-
11unq1() (cypari2.pari_instance.Pari_auto method), 189	listpop
lfunrootres() (cypari2.gen.Gen_base method), 589	_

tres() (cypari2.pari_instance.Pari_auto method), 189 ft() (cypari2.gen.Gen base method), 589 (cypari2.pari_instance.Pari_auto .ft() method), 189 upow() (cypari2.gen.Gen_base method), 589 (cypari2.pari instance.Pari auto ipow() method), 190 eta() (cypari2.gen.Gen_base method), 589 eta() (cypari2.pari_instance.Pari_auto method), 190 etacost() (cypari2.gen.Gen_base method), 590 (cypari2.pari_instance.Pari_auto etacost() method), 190 tainit() (cypari2.gen.Gen_base method), 590 tainit() (cypari2.pari_instance.Pari_auto method), 190 .st() (cypari2.gen.Gen_base method), 591 (cypari2.pari_instance.Pari_auto .st() method), 191 cos() (cypari2.gen.Gen_base method), 591 (cypari2.pari_instance.Pari_auto os() method), 191 (cypari2.gen.Gen base method), 591 (cypari2.pari_instance.Pari_auto method), 192 entered() (cypari2.gen.Gen method), 408 () (cypari2.gen.Gen_base method), 592 () (cypari2.pari_instance.Pari_auto method), 192 () (cypari2.gen.Gen_base method), 592 () (cypari2.pari_instance.Pari_auto_method), 192 () (cypari2.gen.Gen_base method), 592 () (cypari2.pari_instance.Pari_auto method), 192 m() (cypari2.gen.Gen_base method), 592 m() (cypari2.pari instance.Pari auto method), 193 () (cypari2.gen.Gen_base method), 595 () (cypari2.pari_instance.Pari_auto method), 195 (cypari2.gen.Gen method), 408 (cypari2.gen.Gen base method), 425 (cypari2.pari_instance.Pari method), 5 (cypari2.pari_instance.Pari_auto method), 14 sert() (cypari2.gen.Gen_base method), 596 sert() (cypari2.pari_instance.Pari_auto *method*), 196 .1() (cypari2.gen.Gen_base method), 596 .1() (cypari2.pari_instance.Pari_auto method), 196

listpop() (cypari2.gen.Gen_base method), 596

Listpop() (cypari2.pari_instance.Pari_auto method), 196

798

listput() (cypari2.gen.Gen_base method), 596

- listsort() (cypari2.gen.Gen_base method), 597
- listsort() (cypari2.pari_instance.Pari_auto method),
 197
- lngamma() (cypari2.gen.Gen_base method), 597
- lngamma() (cypari2.pari_instance.Pari_auto method),
 198
- localbitprec() (cypari2.gen.Gen_base method), 598
- localbitprec() (cypari2.pari_instance.Pari_auto
 method), 198
- localprec() (cypari2.gen.Gen_base method), 598
- localprec() (cypari2.pari_instance.Pari_auto
 method), 199
- log() (cypari2.gen.Gen_base method), 599
- log() (cypari2.pari_instance.Pari_auto method), 200
- log1p() (cypari2.gen.Gen_base method), 599
- log1p() (cypari2.pari_instance.Pari_auto method), 200
- log_gamma() (cypari2.gen.Gen method), 409
- logint() (cypari2.gen.Gen_base method), 600

Μ

- Map() (cypari2.gen.Gen_base method), 425
- Map() (cypari2.pari_instance.Pari_auto method), 14
- mapdelete() (cypari2.gen.Gen_base method), 600
- mapget() (cypari2.gen.Gen_base method), 601
- mapisdefined() (cypari2.gen.Gen_base method), 601
- mapput() (cypari2.gen.Gen_base method), 601
- Mat() (cypari2.gen.Gen_base method), 425
- Mat() (cypari2.pari_instance.Pari_auto method), 14
- matadjoint() (cypari2.gen.Gen_base method), 602
- matalgtobasis() (cypari2.gen.Gen_base method), 602
- matbasistoalg() (cypari2.gen.Gen_base method), 602
- matcompanion() (cypari2.gen.Gen_base method), 602
- matconcat() (cypari2.gen.Gen_base method), 602

- matconcat() (cypari2.pari_instance.Pari_auto method), 203 matdet() (cypari2.gen.Gen base method), 604 matdet() (cypari2.pari_instance.Pari_auto method), 204 matdetint() (cypari2.gen.Gen base method), 604 matdetint() (cypari2.pari instance.Pari auto method), 204 matdetmod() (cypari2.gen.Gen_base method), 604 matdetmod() (cypari2.pari_instance.Pari_auto method), 204 matdiagonal() (cypari2.gen.Gen_base method), 604 matdiagonal() (cypari2.pari_instance.Pari_auto method), 205 mateigen() (cypari2.gen.Gen_base method), 605 mateigen() (cypari2.pari_instance.Pari_auto method), 205 matfrobenius() (cypari2.gen.Gen base method), 606 matfrobenius() (cypari2.pari_instance.Pari_auto method), 206 mathess() (cypari2.gen.Gen_base method), 606 mathess() (cypari2.pari_instance.Pari_auto method), 206 mathilbert() (cypari2.pari instance.Pari auto method), 206 mathnf() (cypari2.gen.Gen base method), 606
- mathnf() (cypari2.pari_instance.Pari_auto method),
 206
- mathnfmod() (cypari2.gen.Gen_base method), 607
- mathnfmodid() (cypari2.gen.Gen_base method), 608

- matid() (cypari2.pari_instance.Pari_auto method), 209
 matimage() (cypari2.gen.Gen_base method), 609
- matimage() (cypari2.gen.Gen_base melhoa), 609
- matimagecompl() (cypari2.gen.Gen_base method), 609
- matimagemod() (cypari2.gen.Gen_base method), 609
- matindexrank() (cypari2.gen.Gen_base method), 610
- matintersect() (cypari2.gen.Gen_base method), 610

<pre>matinverseimage() (cypari2.gen.Gen_base method), 610</pre>	ma ma
<pre>matinverseimage() (cypari2.pari_instance.Pari_auto</pre>	ma
<pre>matinvmod() (cypari2.gen.Gen base method), 611</pre>	ma
matinymod() (cypari? pari instance Pari auto	
method) 211	ma
method), 211 maticdiagonal () (angri? gan Can base method) 611	ma
matisulagonal() (cypun2.gen.Gen_buse memou), 011	III <i>c</i>
matisdiagonal() (cypari2.pari_instance.Pari_auto method), 212	ma
matker() (cypari2.gen.Gen_base method), 611	ma
matker() (cypari? pari instance Pari auto method)	ma
212	mf
212	
matkerint() (cypari2.gen.Gen meinoa), 409	mı
matkerint() (cypari2.gen.Gen_base method), 611	
<pre>matkerint() (cypari2.pari_instance.Pari_auto</pre>	mf
method), 212	
<pre>matkermod() (cypari2.gen.Gen_base method), 611</pre>	mf
matkermod() (cvpari2.pari instance.Pari auto	
method) 212	
matmuldiagonal() (cynari? gen Gen base method)	mf
612	m1 m4
	mı
matmuldiagonal() (cypari2.pari_instance.Pari_auto	
<i>method</i>), 212	mf
<pre>matmultodiagonal() (cypari2.gen.Gen_base method),</pre>	mf
612	
<pre>matmultodiagonal() (cypari2.pari_instance.Pari_auto</pre>	mf
method), 212	mf
matpascal() (cvpari2.pari_instance.Pari_auto	mf
method) 213	mf
matnermanent() (cupari? gan Gan base method) 612	
matpermanent() (cypaniz.gen. Gen_base memba), 612	d
matpermanent() (cypun2.pan_instance.run_auto	m1
method), 213	mı
matqr() (cypari2.gen.Gen_base method), 612	
<pre>matqr() (cypari2.pari_instance.Pari_auto method), 213</pre>	mf
<pre>matrank() (cypari2.gen.Gen_base method), 613</pre>	mf
<pre>matrank() (cypari2.pari_instance.Pari_auto method),</pre>	
213	mf
matreduce() (cypari2.gen.Gen_base_method), 613	mf
matreduce() (cypari2 pari instance Pari auto	
method) 213	mf
memou), 215	
matrix() (cypari2.pari_instance.Pari methoa), 9	mı
matrixqz() (cypari2.gen.Gen_base method), 613	
matrixqz() (cypari2.pari_instance.Pari_auto method),	mf
214	
<pre>matsize() (cypari2.gen.Gen_base method), 614</pre>	mf
<pre>matsize() (cypari2.pari_instance.Pari_auto method),</pre>	
215	mf
matsnf() (cypari2.gen.Gen. base method), 614	mf
matsnf() (cypani2:geniced_case method)	
215	mı
$\frac{21J}{21}$	
matsolve() (cypani2.gen.Gen_base meinoa), 014	mı
matsolve() (cypari2.pari_instance.Pari_auto method),	
215	mf

atsolvemod() (cypari2.gen.Gen_base method), 615 atsolvemod() (cypari2.pari_instance.Pari_auto method), 215 atsupplement() (cypari2.gen.Gen_base method), 615 atsupplement() (cypari2.pari_instance.Pari_auto method), 216 attranspose() (cypari2.gen.Gen method), 410 attranspose() (cypari2.gen.Gen_base method), 616 attranspose() (cypari2.pari_instance.Pari_auto method), 216 ax() (cypari2.gen.Gen_base method), 616 ax() (cypari2.pari_instance.Pari_auto method), 216 Eatkin() (cypari2.gen.Gen_base method), 616 fatkin() (cypari2.pari_instance.Pari_auto method), 217 fatkineigenvalues() (cypari2.gen.Gen_base method), 617 Eatkineigenvalues() (cvpari2.pari_instance.Pari_auto method), 217 fatkininit() (cypari2.gen.Gen_base method), 617 fatkininit() (cypari2.pari_instance.Pari_auto method), 218 Ebasis() (cypari2.gen.Gen base method), 618 Ebasis() (cypari2.pari_instance.Pari_auto method), 219 Ebd() (cypari2.gen.Gen_base method), 618 Ebd() (cypari2.pari_instance.Pari_auto method), 219 Ebracket() (cypari2.gen.Gen_base method), 618 Ebracket() (cypari2.pari_instance.Pari_auto method), 219 Ecoef() (cypari2.gen.Gen_base method), 618 Ecoef() (cypari2.pari_instance.Pari_auto method), 219 Ecoefs() (cypari2.gen.Gen_base method), 619 Ecoefs() (cypari2.pari_instance.Pari_auto method), 219 Econductor() (cypari2.gen.Gen_base method), 619 Econductor() (cypari2.pari_instance.Pari_auto method), 220 Ecosets() (cypari2.gen.Gen base method), 619 Ecosets() (cypari2.pari_instance.Pari_auto method), 220 Ecuspisregular() (cypari2.gen.Gen_base method), 620 Ecuspisregular() (cypari2.pari_instance.Pari_auto method), 220 Ecusps() (cypari2.gen.Gen_base method), 620 Ecusps() (cypari2.pari_instance.Pari_auto method), 221 Ecuspval() (cypari2.gen.Gen_base method), 620

mfcuspwidth() (cypari2.pari instance.Pari auto method), 221 mfDelta() (cypari2.pari instance.Pari auto method), 216 mfderiv() (cypari2.gen.Gen_base method), 621 mfderiv() (cypari2.pari instance.Pari auto method), 221 mfderivE2() (cypari2.gen.Gen base method), 621 mfderivE2() (cypari2.pari_instance.Pari_auto method), 222 mfdescribe() (cypari2.gen.Gen_base method), 621 mfdescribe() (cypari2.pari_instance.Pari_auto method), 222 mfdim() (cypari2.gen.Gen_base method), 621 mfdim() (cypari2.pari_instance.Pari_auto method), 222 mfdiv() (cypari2.gen.Gen_base method), 622 mfdiv() (cypari2.pari_instance.Pari_auto method), 223 mfEH() (cypari2.gen.Gen base method), 616 mfEH() (cypari2.pari_instance.Pari_auto method), 216 mfeigenbasis() (cypari2.gen.Gen base method), 622 mfeigenbasis() (cypari2.pari_instance.Pari_auto method), 223 mfeigensearch() (cypari2.gen.Gen_base method), 623 (cypari2.pari_instance.Pari_auto mfeigensearch() method), 224 (cypari2.pari_instance.Pari_auto mfeisenstein() method), 225 mfEk() (cypari2.pari_instance.Pari_auto method), 217 mfembed() (cypari2.gen.Gen_base method), 624 mfembed() (cypari2.pari_instance.Pari_auto method), 225 mfeval() (cypari2.gen.Gen_base method), 625 mfeval() (cypari2.pari_instance.Pari_auto method), 227 mffields() (cypari2.gen.Gen base method), 625 mffields() (cypari2.pari_instance.Pari_auto method), 227 mffromell() (cypari2.gen.Gen_base method), 626 mffromell() (cypari2.pari_instance.Pari_auto method), 228 mffrometaquo() (cypari2.gen.Gen base method), 626 mffrometaquo() (cypari2.pari_instance.Pari_auto method), 228 mffromlfun() (cypari2.gen.Gen_base method), 627 mffromlfun() (cypari2.pari_instance.Pari_auto method), 228 mffromqf() (cypari2.gen.Gen_base method), 627 mffromqf() (cypari2.pari_instance.Pari_auto method), 229 mfgaloisprojrep() (cypari2.gen.Gen_base method), 628 mfgaloisprojrep() (cypari2.pari_instance.Pari_auto method), 229 mfgaloistype() (cypari2.gen.Gen base method), 629

mfgaloistype() (cypari2.pari_instance.Pari_auto method), 230 mfhecke() (cypari2.gen.Gen base method), 629 mfhecke() (cypari2.pari_instance.Pari_auto method), 231 mfheckemat() (cypari2.gen.Gen base method), 630 mfheckemat() (cypari2.pari instance.Pari auto method), 231 mfinit() (cypari2.gen.Gen base method), 630 mfinit() (cypari2.pari_instance.Pari_auto method), 232 mfisCM() (cypari2.gen.Gen_base method), 631 mfisCM() (cypari2.pari_instance.Pari_auto method), 232 mfisequal() (cypari2.gen.Gen_base method), 631 mfisequal() (cypari2.pari_instance.Pari_auto method), 232 mfisetaquo() (cypari2.gen.Gen base method), 631 mfisetaquo() (cypari2.pari_instance.Pari_auto method), 233 mfkohnenbasis() (cypari2.gen.Gen_base method), 631 mfkohnenbasis() (cypari2.pari_instance.Pari_auto method), 233 mfkohnenbijection() (cypari2.gen.Gen base *method*), 632 mfkohnenbijection() (cypari2.pari_instance.Pari_auto method), 233 mfkohneneigenbasis() (cypari2.gen.Gen_base method), 633 mfkohneneigenbasis() (cvpari2.pari_instance.Pari_auto method), 235 mflinear() (cypari2.gen.Gen_base method), 634 mflinear() (cypari2.pari_instance.Pari_auto method), 235 mfmanin() (cypari2.gen.Gen base method), 634 mfmanin() (cypari2.pari_instance.Pari_auto method), 236 mfmul() (cypari2.gen.Gen_base method), 635 mfmul() (cypari2.pari instance.Pari auto method), 236 mfnumcusps() (cypari2.gen.Gen base method), 635 (cypari2.pari instance.Pari auto mfnumcusps() method), 236 mfparams() (cypari2.gen.Gen_base method), 635 mfparams() (cypari2.pari_instance.Pari_auto method), 236 mfperiodpol() (cypari2.gen.Gen_base method), 635 mfperiodpol() (cypari2.pari_instance.Pari_auto method), 237 mfperiodpolbasis() (cypari2.pari_instance.Pari_auto method), 237 mfpetersson() (cypari2.gen.Gen_base method), 636 (cypari2.pari instance.Pari auto mfpetersson()

method), 237 mfpow() (cypari2.gen.Gen_base method), 637 mfpow() (cypari2.pari instance.Pari auto method), 238 mfsearch() (cypari2.gen.Gen_base method), 637 mfsearch() (cypari2.pari instance.Pari auto method), 238 mfshift() (cypari2.gen.Gen base method), 637 mfshift() (cypari2.pari_instance.Pari_auto method), 239 mfshimura() (cypari2.gen.Gen_base method), 637 (cypari2.pari_instance.Pari_auto mfshimura() method), 239 mfslashexpansion() (cypari2.gen.Gen_base method), 638 mfslashexpansion() (cypari2.pari_instance.Pari_auto method), 240 mfspace() (cypari2.gen.Gen_base method), 639 mfspace() (cypari2.pari_instance.Pari_auto method), 241 mfsplit() (cypari2.gen.Gen base method), 640 mfsplit() (cypari2.pari_instance.Pari_auto method), 241 mfsturm() (cypari2.gen.Gen_base method), 640 mfsturm() (cypari2.pari instance.Pari auto method), 242 mfsymbol() (cypari2.gen.Gen_base method), 641 mfsymbol() (cypari2.pari_instance.Pari_auto method), 242 mfsymboleval() (cypari2.gen.Gen_base method), 641 mfsymboleval() (cypari2.pari_instance.Pari_auto method), 243 mftaylor() (cypari2.gen.Gen_base method), 642 mftaylor() (cypari2.pari_instance.Pari_auto method), 244 mfTheta() (cypari2.gen.Gen base method), 616 mfTheta() (cypari2.pari_instance.Pari_auto method), 217 mftobasis() (cypari2.gen.Gen_base method), 642 mftobasis() (cypari2.pari_instance.Pari_auto method), 244 mftocoset() (cypari2.pari_instance.Pari_auto method), 245 mftonew() (cypari2.gen.Gen base method), 643 mftonew() (cypari2.pari_instance.Pari_auto method), 245 mftraceform() (cypari2.gen.Gen_base method), 643 (cypari2.pari_instance.Pari_auto mftraceform() method), 245 mftwist() (cypari2.gen.Gen_base method), 644 mftwist() (cypari2.pari_instance.Pari_auto method), 245 min() (cypari2.gen.Gen_base method), 644 min() (cypari2.pari instance.Pari auto method), 246 minpoly() (cypari2.gen.Gen base method), 644

minpoly() (cypari2.pari_instance.Pari_auto method), 246 mod() (cypari2.gen.Gen method), 410 Mod() (cypari2.gen.Gen_base method), 426 Mod() (cypari2.pari_instance.Pari_auto method), 15 modreverse() (cypari2.gen.Gen base method), 644 modreverse() (cypari2.pari instance.Pari auto method), 246 module cypari2.closure, 769 cypari2.convert,774 cypari2.gen, 382 cypari2.handle_error, 772 cypari2.pari_instance, 1 cypari2.stack,768 moebius() (cypari2.gen.Gen_base method), 645 moebius() (cypari2.pari_instance.Pari_auto method), 246 msatkinlehner() (cypari2.gen.Gen_base method), 645 msatkinlehner() (cypari2.pari instance.Pari auto method), 246 mscosets() (cypari2.gen.Gen_base method), 645 mscosets() (cypari2.pari_instance.Pari_auto method), 247 mscuspidal() (cypari2.gen.Gen_base method), 645 mscuspidal() (cypari2.pari_instance.Pari_auto method), 247 msdim() (cypari2.gen.Gen_base method), 646 msdim() (cypari2.pari_instance.Pari_auto method), 248 mseisenstein() (cypari2.gen.Gen_base method), 646 mseisenstein() (cypari2.pari_instance.Pari_auto method), 248 mseval() (cypari2.gen.Gen_base method), 647 mseval() (cypari2.pari_instance.Pari_auto method), 248 msfarey() (cypari2.gen.Gen_base method), 647 msfarey() (cypari2.pari instance.Pari auto method), 249 msfromcusp() (cypari2.gen.Gen_base method), 648 msfromcusp() (cypari2.pari_instance.Pari_auto method), 249 msfromell() (cypari2.gen.Gen base method), 649 msfromell() (cypari2.pari_instance.Pari_auto method), 250 msfromhecke() (cypari2.gen.Gen_base method), 650 msfromhecke() (cypari2.pari_instance.Pari_auto method), 251 msgetlevel() (cypari2.gen.Gen_base method), 650 msgetlevel() (cypari2.pari_instance.Pari_auto method), 252 msgetsign() (cypari2.gen.Gen_base method), 650 (cypari2.pari_instance.Pari_auto msgetsign() method), 252 msgetweight() (cypari2.gen.Gen base method), 650

- mshecke() (cypari2.gen.Gen_base method), 650
- msinit() (cypari2.gen.Gen_base method), 651
- msissymbol() (cypari2.gen.Gen_base method), 652
- msissymbol() (cypari2.pari_instance.Pari_auto method), 253
- mslattice() (cypari2.gen.Gen_base method), 652
- msnew() (cypari2.gen.Gen_base method), 653
- msnew() (cypari2.pari_instance.Pari_auto method), 255
- msomseval() (cypari2.gen.Gen_base method), 653
- mspadicinit() (cypari2.gen.Gen_base method), 655
- mspadicL() (cypari2.gen.Gen_base method), 653

- mspadicseries() (cypari2.gen.Gen_base method), 656
- mspathgens() (cypari2.gen.Gen_base method), 657
- mspathlog() (cypari2.gen.Gen_base method), 658
- mspetersson() (cypari2.gen.Gen_base method), 659
- mspolygon() (cypari2.gen.Gen_base method), 660
- msqexpansion() (cypari2.gen.Gen_base method), 661
- mssplit() (cypari2.gen.Gen_base method), 662
- msstar() (cypari2.gen.Gen_base method), 662
- mstooms() (cypari2.gen.Gen_base method), 662

Ν

- ncols() (cypari2.gen.Gen method), 410
- newtonpoly() (cypari2.gen.Gen_base method), 663
- nextprime() (cypari2.gen.Gen method), 411
- nextprime() (cypari2.gen.Gen_base method), 663
- nf_get_diff() (cypari2.gen.Gen method), 411
- nf_get_pol() (cypari2.gen.Gen method), 411
- nf_get_sign() (cypari2.gen.Gen method), 412
- nf_get_zk() (cypari2.gen.Gen method), 412
- nf_subst() (cypari2.gen.Gen method), 413
- nfalgtobasis() (cypari2.gen.Gen_base method), 663
- nfbasis() (cypari2.gen.Gen method), 413
- nfbasis() (cypari2.gen.Gen_base method), 664
- nfbasis() (cypari2.pari_instance.Pari_auto method), 265
- nfbasis_d() (cypari2.gen.Gen method), 414
- nfbasistoalg() (cypari2.gen.Gen_base method), 666
- nfbasistoalg_lift() (cypari2.gen.Gen method), 415
- nfcertify() (cypari2.gen.Gen_base method), 666
- nfcertify() (cypari2.pari_instance.Pari_auto
 method), 267
- nfcompositum() (cypari2.gen.Gen_base method), 666
- nfdetint() (cypari2.gen.Gen_base method), 667
- nfdisc() (cypari2.gen.Gen_base method), 667
- nfdisc() (cypari2.pari_instance.Pari_auto method), 269
- nfdiscfactors() (cypari2.gen.Gen_base method), 668
- nfdiscfactors() (cypari2.pari_instance.Pari_auto method), 269
- nfeltadd() (cypari2.gen.Gen_base method), 668
- nfeltdiv() (cypari2.gen.Gen_base method), 668

nfeltdiveuc() (cypari2.gen.Gen_base method), 668 nfeltdiveuc() (cypari2.pari instance.Pari auto method), 270 nfeltdivmodpr() (cypari2.gen.Gen_base method), 668 nfeltdivmodpr() (cypari2.pari_instance.Pari_auto method), 270 nfeltdivrem() (cypari2.gen.Gen base method), 669 (cypari2.pari_instance.Pari_auto nfeltdivrem() method), 270 nfeltembed() (cypari2.gen.Gen_base method), 669 nfeltembed() (cypari2.pari_instance.Pari_auto method), 270 nfeltmod() (cypari2.gen.Gen_base method), 669 nfeltmod() (cypari2.pari_instance.Pari_auto method), 271nfeltmul() (cypari2.gen.Gen_base method), 669 nfeltmul() (cypari2.pari_instance.Pari_auto method), 271 nfeltmulmodpr() (cypari2.gen.Gen_base method), 669 (cypari2.pari_instance.Pari_auto nfeltmulmodpr() method), 271 nfeltnorm() (cypari2.gen.Gen_base method), 669 (cypari2.pari_instance.Pari_auto nfeltnorm() method), 271 nfeltpow() (cypari2.gen.Gen_base method), 669 nfeltpow() (cypari2.pari instance.Pari auto method), 271nfeltpowmodpr() (cypari2.gen.Gen_base method), 669 nfeltpowmodpr() (cypari2.pari_instance.Pari_auto method), 271 nfeltreduce() (cypari2.gen.Gen_base method), 669 nfeltreduce() (cypari2.pari_instance.Pari_auto method), 271 nfeltreducemodpr() (cypari2.gen.Gen_base method), 670 nfeltreducemodpr() (cypari2.pari instance.Pari auto method), 271 nfeltsign() (cypari2.gen.Gen_base method), 670 nfeltsign() (cypari2.pari_instance.Pari_auto method), 271 nfelttrace() (cypari2.gen.Gen base method), 670 nfelttrace() (cypari2.pari_instance.Pari_auto method). 272 nfeltval() (cypari2.gen.Gen method), 415 nfeltval() (cypari2.gen.Gen_base method), 670 nfeltval() (cypari2.pari_instance.Pari_auto method), 272 nffactor() (cypari2.gen.Gen_base method), 671 nffactor() (cypari2.pari_instance.Pari_auto method), 272 nffactorback() (cypari2.gen.Gen_base method), 671 nffactorback() (cypari2.pari instance.Pari auto method), 273

nffactormod() (cypari2.gen.Gen_base method), 671

nfgaloisapply() (*cypari2.gen.Gen_base method*), 672

nfgaloisconj() (cypari2.gen.Gen_base method), 672

nfgaloisconj() (cypari2.pari_instance.Pari_auto method), 274

- nfgrunwaldwang() (cypari2.gen.Gen_base method), 673
- nfhilbert() (cypari2.gen.Gen_base method), 673
- nfhnf() (cypari2.gen.Gen_base method), 674
- nfhnf() (cypari2.pari_instance.Pari_auto method), 275
- nfhnfmod() (cypari2.gen.Gen_base method), 674
- nfhnfmod() (cypari2.pari_instance.Pari_auto method), 275
- nfinit() (cypari2.gen.Gen_base method), 674
- nfisideal() (cypari2.gen.Gen_base method), 676
- nfisideal() (cypari2.pari_instance.Pari_auto
 method), 277
- nfisincl() (cypari2.gen.Gen_base method), 676
- nfisincl() (cypari2.pari_instance.Pari_auto method), 277
- nfisisom() (cypari2.gen.Gen_base method), 676
- nfisisom() (cypari2.pari_instance.Pari_auto method), 278

- nfkermodpr() (cypari2.gen.Gen_base method), 678
- nfmodpr() (cypari2.gen.Gen_base method), 678

nfmodpr() (cypari2.pari_instance.Pari_auto method), 279

- nfmodprinit() (cypari2.gen.Gen_base method), 678
- nfmodprlift() (cypari2.gen.Gen_base method), 679
- nfnewprec() (cypari2.gen.Gen_base method), 679
- nfnewprec() (cypari2.pari_instance.Pari_auto method), 281
- nfpolsturm() (cypari2.gen.Gen_base method), 679
- nfpolsturm() (cypari2.pari_instance.Pari_auto method), 281

nfroots() (cypari2.gen.Gen base method), 680

- nfroots() (cypari2.pari_instance.Pari_auto method), 281
- nfrootsof1() (cypari2.gen.Gen_base method), 680
- (cypari2.pari_instance.Pari_auto nfrootsof1() method), 282
- nfsnf() (cypari2.gen.Gen base method), 680
- nfsnf() (cypari2.pari_instance.Pari_auto method), 282
- nfsolvemodpr() (cypari2.gen.Gen_base method), 681
- nfsolvemodpr() (cypari2.pari_instance.Pari_auto method), 282
- nfsplitting() (cypari2.gen.Gen_base method), 681
- (cypari2.pari_instance.Pari_auto nfsplitting() method), 282
- nfsubfields() (cypari2.gen.Gen_base method), 681
- nfsubfields() (cypari2.pari_instance.Pari_auto method), 283
- nfsubfieldscm() (cypari2.gen.Gen base method), 682
- nfsubfieldscm() (cypari2.pari_instance.Pari_auto method), 284
- nfsubfieldsmax() (cypari2.gen.Gen_base method), 682
- nfsubfieldsmax() (cypari2.pari_instance.Pari_auto method), 284
- norm() (cypari2.gen.Gen_base method), 682
- norm() (cypari2.pari instance.Pari auto method), 284
- norml2() (cypari2.gen.Gen_base method), 682
- norm12() (cypari2.pari_instance.Pari_auto method), 284
- normlp() (cypari2.gen.Gen_base method), 683
- normlp() (cypari2.pari_instance.Pari_auto method), 284
- nrows() (cypari2.gen.Gen method), 415
- numbpart() (cypari2.gen.Gen_base method), 683
- numbpart() (cypari2.pari instance.Pari auto method), 285
- numdiv() (cypari2.gen.Gen base method), 683
- numdiv() (cypari2.pari_instance.Pari_auto method), 285
- numerator() (cypari2.gen.Gen_base method), 683
- numerator() (cypari2.pari instance.Pari auto method), 285
- numtoperm() (cypari2.pari_instance.Pari_auto method), 285

Ο

objtoclosure() (in module cypari2.closure), 771 objtogen() (in module cypari2.gen), 766 omega() (cypari2.gen.Gen method), 415 omega() (cypari2.gen.Gen_base method), 684 omega() (cypari2.pari_instance.Pari_auto method), 285 one() (cypari2.pari_instance.Pari method), 9 oo() (cypari2.pari_instance.Pari_auto method), 286

Ρ

padicappr() (cypari2.gen.Gen base method), 684 padicappr() (cypari2.pari instance.Pari auto method), 286 padicfields() (cypari2.gen.Gen_base method), 684 padicfields() (cypari2.pari_instance.Pari_auto method), 286 padicprec() (cypari2.gen.Gen_base method), 684 (cypari2.pari_instance.Pari_auto padicprec() method), 286 padicprime() (cypari2.gen.Gen method), 416 parapply() (cypari2.gen.Gen_base method), 685 parapply() (cypari2.pari_instance.Pari_auto method), 287 pareval() (cypari2.gen.Gen base method), 685 pareval() (cypari2.pari instance.Pari auto method), 287 Pari (class in cypari2.pari_instance), 5 Pari_auto (class in cypari2.pari_instance), 13 pari_version() (cypari2.pari_instance.Pari static method), 9 PariError, 773 parselect() (cypari2.gen.Gen_base method), 685 (cypari2.pari_instance.Pari_auto parselect() method), 287 partitions() (cypari2.pari_instance.Pari_auto method), 287 permcycles() (cypari2.gen.Gen_base method), 686 permcycles() (cypari2.pari_instance.Pari_auto method), 288 permorder() (cypari2.gen.Gen_base method), 686 permorder() (cypari2.pari instance.Pari auto method), 288 permsign() (cypari2.gen.Gen_base method), 686 permsign() (cypari2.pari_instance.Pari_auto method), 288 permtonum() (cypari2.gen.Gen_base method), 686 permtonum() (cypari2.pari_instance.Pari_auto method), 289 pi() (cypari2.pari_instance.Pari method), 9 Pi() (cypari2.pari_instance.Pari_auto method), 16 plotbox() (cypari2.pari_instance.Pari_auto method), 289 plotclip() (cypari2.pari_instance.Pari_auto method), 289 plotcolor() (cypari2.pari_instance.Pari_auto method), 289 plotcopy() (cypari2.pari_instance.Pari_auto method), 289 plotcursor() (cypari2.pari instance.Pari auto method), 289 plotdraw() (cypari2.gen.Gen_base method), 686

plotdraw() (cypari2.pari_instance.Pari_auto method), 289

- plotexport() (cypari2.gen.Gen_base method), 686
 plotexport() (cypari2.pari_instance.Pari_auto
- method), 290
- plothraw() (cypari2.gen.Gen_base method), 686

- plothsizes() (cypari2.pari_instance.Pari_auto method), 290

- plotstring() (cypari2.pari_instance.Pari_auto method), 292
- Pol() (cypari2.gen.Gen_base method), 427
- Pol() (cypari2.pari_instance.Pari_auto method), 16

- polclass() (cypari2.gen.Gen_base method), 687
- polcoef() (cypari2.gen.Gen_base method), 688

- polcoeff() (cypari2.gen.Gen_base method), 688
- polcompositum() (cypari2.gen.Gen_base method), 688

- poldegree() (cypari2.gen.Gen_base method), 690
- poldisc() (cypari2.gen.Gen_base method), 690

- polgalois() (cypari2.gen.Gen_base method), 691
- polgraeffe() (cypari2.gen.Gen_base method), 691
- polhensellift() (cypari2.gen.Gen_base method), 692
- polhermite() (cypari2.pari_instance.Pari_auto method), 298
- polinterpolate() (cypari2.gen.Gen method), 416

- poliscyclo() (cypari2.gen.Gen_base method), 694
- poliscyclo() (cypari2.pari_instance.Pari_auto method), 300

- pollaguerre() (cypari2.pari_instance.Pari_auto

method), 300

- pollead() (cypari2.gen.Gen_base method), 694

- polrecip() (cypari2.gen.Gen_base method), 694
- polred() (cypari2.gen.Gen method), 416
- polred() (cypari2.gen.Gen_base method), 694
- polredabs() (cypari2.gen.Gen_base method), 695
- polredbest() (cypari2.gen.Gen_base method), 696
- polredord() (cypari2.gen.Gen_base method), 697
- polresultant() (cypari2.gen.Gen_base method), 697

- Polrev() (cypari2.gen.Gen_base method), 427
- Polrev() (cypari2.pari_instance.Pari_auto method), 16
- polroots() (cypari2.gen.Gen_base method), 697
- polrootsbound() (cypari2.gen.Gen_base method), 698
- polrootsff() (cypari2.gen.Gen_base method), 698
- polrootsmod() (cypari2.gen.Gen_base method), 698
- polrootspadic() (cypari2.gen.Gen_base method), 699
- polrootsreal() (cypari2.gen.Gen_base method), 699
- polsturm() (cypari2.gen.Gen_base method), 700

- polsubcyclo() (cypari2.pari_instance.Pari method), 9 polsubcyclo() (cypari2.pari instance.Pari auto method), 307 polsylvestermatrix() (cypari2.gen.Gen_base method), 700 polsylvestermatrix() (cypari2.pari instance.Pari auto method), 307 polsym() (cypari2.gen.Gen_base method), 700 polsym() (cypari2.pari_instance.Pari_auto method), 307 poltchebi() (cypari2.pari_instance.Pari_auto method), 308 polteichmuller() (cypari2.gen.Gen_base method), 700 polteichmuller() (cypari2.pari_instance.Pari_auto method), 308 poltschirnhaus() (cypari2.gen.Gen_base method), 701 poltschirnhaus() (cypari2.pari instance.Pari auto method), 308 polylog() (cypari2.gen.Gen method), 416 polylog() (cypari2.pari_instance.Pari_auto method), 308 polylogmult() (cypari2.gen.Gen_base method), 701 (cypari2.pari_instance.Pari_auto polylogmult() method), 308 polzagier() (cypari2.pari_instance.Pari_auto method), 309 powers() (cypari2.gen.Gen_base method), 701 powers() (cypari2.pari_instance.Pari_auto method), 309 pr_get_e() (cypari2.gen.Gen method), 417 pr_get_f() (cypari2.gen.Gen method), 417 pr_get_gen() (cvpari2.gen.Gen method), 418 pr_get_p() (cypari2.gen.Gen method), 418 prec_bits_to_dec() (in module cypari2.pari_instance), 380 prec_bits_to_words() module (in cvpari2.pari_instance), 381 prec_dec_to_bits() (in module cypari2.pari_instance), 381
- prec_dec_to_words() (in module cypari2.pari_instance), 381 prec_words_to_bits() (in module cy-
- prec_words_to_bits() (in module cypari2.pari_instance), 382 prec_words_to_dec() (in module cy-
- prec_words_co_dec() (in module cy pari2.pari_instance), 382
- precision() (cypari2.gen.Gen_base method), 702
- precprime() (cypari2.gen.Gen_base method), 702

- prime() (cypari2.pari_instance.Pari_auto method), 310 primecert() (cypari2.gen.Gen_base method), 702 primecert() (cypari2.pari_instance.Pari_auto method), 310 primecertexport() (cypari2.gen.Gen_base method), 703 primecertexport() (cypari2.pari instance.Pari auto *method*), 311primecertisvalid() (cypari2.gen.Gen_base method), 704 primecertisvalid() (cypari2.pari_instance.Pari_auto method), 312 primepi() (cypari2.gen.Gen_base method), 705 primepi() (cypari2.pari_instance.Pari_auto method), 313 primes() (cypari2.gen.Gen_base method), 705 primes() (cypari2.pari_instance.Pari method), 10 primes() (cypari2.pari_instance.Pari_auto method), 313 prodeulerrat() (cypari2.gen.Gen base method), 705 prodeulerrat() (cypari2.pari_instance.Pari_auto method), 313 prodnumrat() (cypari2.gen.Gen_base method), 705 (cypari2.pari_instance.Pari_auto prodnumrat() method), 313 psdraw() (cypari2.gen.Gen_base method), 705 psdraw() (cypari2.pari_instance.Pari_auto method), 313 psi() (cypari2.gen.Gen_base method), 705 psi() (cypari2.pari_instance.Pari_auto method), 313 psplothraw() (cypari2.gen.Gen_base method), 705 psplothraw() (cypari2.pari_instance.Pari_auto method), 313 python() (cypari2.gen.Gen method), 418 python_list() (cvpari2.gen.Gen method), 418 python_list_small() (cypari2.gen.Gen method), 419 Q qfauto() (cypari2.gen.Gen_base method), 706 qfauto() (cypari2.pari_instance.Pari_auto method), 314 qfautoexport() (cypari2.gen.Gen_base method), 706 qfautoexport() (cypari2.pari_instance.Pari_auto method), 314 Qfb() (cypari2.gen.Gen base method), 427 Qfb() (cypari2.pari_instance.Pari_auto method), 17 qfbclassno() (cypari2.gen.Gen_base method), 706 (cypari2.pari_instance.Pari_auto qfbclassno() method), 314 qfbcompraw() (cypari2.gen.Gen base method), 707
- qfbhclassno() (cypari2.gen.Gen_base method), 707
- gfbhclassno() (cypari2.pari_instance.Pari_auto method), 315 qfbil() (cypari2.gen.Gen base method), 707 qfbil() (cypari2.pari_instance.Pari_auto method), 315 qfbnucomp() (cypari2.gen.Gen_base method), 707 qfbnucomp() (cypari2.pari instance.Pari auto method), 315 qfbnupow() (cypari2.gen.Gen_base method), 707 qfbnupow() (cypari2.pari_instance.Pari_auto method), 315 qfbpowraw() (cypari2.gen.Gen_base method), 707 qfbpowraw() (cypari2.pari_instance.Pari_auto method), 315 qfbprimeform() (cypari2.gen.Gen_base method), 707 qfbprimeform() (cypari2.pari_instance.Pari_auto method), 315 qfbred() (cypari2.gen.Gen_base method), 708 qfbred() (cypari2.pari_instance.Pari_auto method), 316 qfbreds12() (cypari2.gen.Gen base method), 708 qfbreds12() (cypari2.pari_instance.Pari_auto method), 316 qfbsolve() (cypari2.gen.Gen_base method), 708 qfbsolve() (cypari2.pari instance.Pari auto method), 316 qfeval() (cypari2.gen.Gen_base method), 709 qfeval() (cypari2.pari_instance.Pari_auto method), 317 qfgaussred() (cypari2.gen.Gen_base method), 710 qfgaussred() (cypari2.pari_instance.Pari_auto method), 318 qfisom() (cypari2.gen.Gen_base method), 711 qfisom() (cypari2.pari_instance.Pari_auto method), 319 gfisominit() (cypari2.gen.Gen base method), 711 qfisominit() (cypari2.pari_instance.Pari_auto method), 319 qfjacobi() (cypari2.gen.Gen_base method), 711 qfjacobi() (cypari2.pari_instance.Pari_auto method), 319 qflll() (cypari2.gen.Gen base method), 712 qflll() (cypari2.pari_instance.Pari_auto method), 320 qflllgram() (cypari2.gen.Gen_base method), 712 qflllgram() (cypari2.pari_instance.Pari_auto method), 320 qfminim() (cypari2.gen.Gen_base method), 713 qfminim() (cypari2.pari_instance.Pari_auto method), 321 qfnorm() (cypari2.gen.Gen_base method), 714 qfnorm() (cypari2.pari_instance.Pari_auto method), 322
- qforbits() (cypari2.gen.Gen_base method), 714

- qfparam() (cypari2.gen.Gen_base method), 714
- qfperfection() (cypari2.gen.Gen_base method), 715
- qfrep() (cypari2.gen.Gen method), 419
- qfrep() (cypari2.gen.Gen_base method), 715
- qfrep() (cypari2.pari_instance.Pari_auto method), 323
- qfsign() (cypari2.gen.Gen_base method), 715
- qfsolve() (cypari2.gen.Gen_base method), 715
- quadclassunit() (cypari2.gen.Gen_base method), 716
- quaddisc() (cypari2.gen.Gen_base method), 716
- quadgen() (cypari2.gen.Gen_base method), 716
- quadhilbert() (cypari2.gen.Gen_base method), 717
- quadpoly() (cypari2.gen.Gen_base method), 717
- quadray() (cypari2.gen.Gen_base method), 717
- quadray() (cypari2.pari_instance.Pari_auto method), 325
- quadregulator() (cypari2.gen.Gen_base method), 717
 quadregulator() (cypari2.pari_instance.Pari_auto
- *method*), 325
- quadunit() (cypari2.gen.Gen_base method), 717

R

- ramanujantau() (cypari2.gen.Gen_base method), 717
- random() (cypari2.gen.Gen_base method), 718
- randomprime() (cypari2.gen.Gen_base method), 719
- read() (cypari2.pari_instance.Pari_auto method), 327

- readvec() (cypari2.pari_instance.Pari_auto method), 328
- real() (cypari2.gen.Gen_base method), 719
- real() (cypari2.pari_instance.Pari_auto method), 328
- removeprimes() (cypari2.gen.Gen_base method), 720
- rnfalgtobasis() (cypari2.gen.Gen_base method), 720
- rnfbasis() (cypari2.gen.Gen_base method), 720
- rnfbasistoalg() (cypari2.gen.Gen_base method), 720
- rnfcharpoly() (cypari2.gen.Gen_base method), 720
- rnfconductor() (cypari2.gen.Gen_base method), 720
- rnfdedekind() (cypari2.gen.Gen_base method), 720
- rnfdet() (cypari2.gen.Gen_base method), 721
- rnfdisc() (cypari2.gen.Gen_base method), 721
- rnfeltabstorel() (cypari2.gen.Gen_base method),
 722
- rnfeltdown() (cypari2.gen.Gen_base method), 722
- rnfeltnorm() (cypari2.gen.Gen_base method), 723
- rnfeltreltoabs() (cypari2.gen.Gen_base method),
 723
- rnfelttrace() (cypari2.gen.Gen_base method), 723
- rnfeltup() (cypari2.gen.Gen_base method), 723
- rnfequation() (cypari2.gen.Gen_base method), 724

rnfhnfbasis() (cypari2.gen.Gen_base method), 724

rnfhnfbasis() (cypari2.pari_instance.Pari_auto method), 333

- rnfidealabstorel() (cypari2.gen.Gen_base method),
 724
- rnfidealdown() (cypari2.gen.Gen_base method), 725

- rnfidealhnf() (cypari2.gen.Gen_base method), 725
- rnfidealmul() (cypari2.gen.Gen_base method), 726
- rnfidealnormabs() (cypari2.gen.Gen_base method),
 726
- rnfidealnormrel() (cypari2.gen.Gen_base method),
 726
- rnfidealprimedec() (cypari2.gen.Gen_base method),
 726
- rnfidealreltoabs() (cypari2.gen.Gen_base method),
 726
- rnfidealtwoelt() (cypari2.gen.Gen_base method),
 727
- rnfidealup() (cypari2.gen.Gen_base method), 727
- rnfinit() (cypari2.gen.Gen_base method), 727
- rnfisabelian() (cypari2.gen.Gen_base method), 728
- rnfisfree() (cypari2.gen.Gen_base method), 729
- rnfislocalcyclo() (cypari2.gen.Gen_base method),
 729

- rnfisnorm() (cypari2.gen.Gen_base method), 729
- rnfisnorminit() (cypari2.gen.Gen_base method), 729
- rnfkummer() (cypari2.gen.Gen_base method), 730
- rnflllgram() (cypari2.gen.Gen_base method), 730
- rnfnormgroup() (cypari2.gen.Gen_base method), 730
- rnfpolred() (cypari2.gen.Gen_base method), 730
- rnfpolredabs() (cypari2.gen.Gen_base method), 730
- rnfpolredbest() (cypari2.gen.Gen_base method), 730
- rnfpseudobasis() (cypari2.gen.Gen_base method),
 731
- rnfsteinitz() (cypari2.gen.Gen_base method), 732

- round() (cypari2.gen.Gen method), 420
- round() (cypari2.gen.Gen_base method), 732
- round() (cypari2.pari_instance.Pari_auto method), 341

S

- sage() (cypari2.gen.Gen method), 420
- select() (cypari2.gen.Gen_base method), 732
- self() (cypari2.pari_instance.Pari_auto method), 342
- Ser() (cypari2.gen.Gen method), 383
- Ser() (cypari2.gen.Gen_base method), 427
- Ser() (cypari2.pari_instance.Pari_auto method), 17
- seralgdep() (cypari2.gen.Gen_base method), 733
- seralgdep() (cypari2.pari_instance.Pari_auto
 method), 342
- serchop() (cypari2.gen.Gen_base method), 734

serconvol() (cypari2.gen.Gen_base method), 734 serconvol() (cypari2.pari instance.Pari auto method), 343 serlaplace() (cypari2.gen.Gen_base method), 734 serlaplace() (cypari2.pari_instance.Pari_auto method), 343 serprec() (cypari2.gen.Gen base method), 734 serprec() (cypari2.pari_instance.Pari_auto method), 343 serreverse() (cypari2.gen.Gen_base method), 734 serreverse() (cypari2.pari_instance.Pari_auto method), 343 Set() (cypari2.gen.Gen_base method), 428 Set() (cypari2.pari_instance.Pari_auto method), 18 set_debug_level() (cypari2.pari_instance.Pari method), 11 set_real_precision() (cypari2.pari_instance.Pari method), 11 set_real_precision_bits() (cypari2.pari instance.Pari method), 11 setbinop() (cypari2.gen.Gen_base method), 734 setbinop() (cypari2.pari_instance.Pari_auto method), 343 setintersect() (cypari2.gen.Gen base method), 735 (cypari2.pari_instance.Pari_auto setintersect() method), 344 setisset() (cypari2.gen.Gen_base method), 735 setisset() (cypari2.pari_instance.Pari_auto method), 344 setminus() (cypari2.gen.Gen_base method), 735 setminus() (cypari2.pari_instance.Pari_auto method), 344 setrand() (cypari2.gen.Gen_base method), 735 setrand() (cypari2.pari_instance.Pari method), 12 setrand() (cypari2.pari_instance.Pari_auto method), 344 setsearch() (cypari2.gen.Gen base method), 735 setsearch() (cypari2.pari_instance.Pari_auto method), 344 setunion() (cypari2.gen.Gen_base method), 736 setunion() (cypari2.pari instance.Pari auto method), 345 shift() (cypari2.gen.Gen base method), 736 shift() (cypari2.pari_instance.Pari_auto method), 345 shiftmul() (cypari2.gen.Gen_base method), 736 shiftmul() (cypari2.pari_instance.Pari_auto method), 345 sigma() (cypari2.gen.Gen_base method), 736 sigma() (cypari2.pari_instance.Pari_auto method), 345 sign() (cypari2.gen.Gen_base method), 736 sign() (cypari2.pari_instance.Pari_auto method), 345 simplify() (cypari2.gen.Gen_base method), 736 simplify() (cypari2.pari_instance.Pari_auto method),

sin() (cvpari2.gen.Gen base method), 737 sin() (cypari2.pari_instance.Pari_auto method), 346 sinc() (cypari2.gen.Gen base method), 737 sinc() (cypari2.pari_instance.Pari_auto method), 346 sinh() (cypari2.gen.Gen base method), 737 sinh() (cypari2.pari instance.Pari auto method), 346 sizebyte() (cypari2.gen.Gen method), 420 sizebyte() (cypari2.gen.Gen_base method), 737 sizebyte() (cypari2.pari instance.Pari auto method), 346 sizedigit() (cypari2.gen.Gen_base method), 737 sizedigit() (cypari2.pari_instance.Pari_auto method), 346 sizeword() (cypari2.gen.Gen method), 421 sqr() (cypari2.gen.Gen_base method), 737 sqr() (cypari2.pari_instance.Pari_auto method), 346 sqrt() (cypari2.gen.Gen_base method), 738 sqrt() (cypari2.pari instance.Pari auto method), 347 sqrtint() (cypari2.gen.Gen_base method), 738 sqrtint() (cypari2.pari instance.Pari auto method), 347 sqrtn() (cypari2.gen.Gen method), 421 sqrtn() (cypari2.gen.Gen_base method), 738 sqrtn() (cypari2.pari instance.Pari auto method), 347 sqrtnint() (cypari2.gen.Gen base method), 739 sqrtnint() (cypari2.pari_instance.Pari_auto method), 348 stacksize() (cypari2.pari_instance.Pari method), 12 stacksizemax() (cypari2.pari_instance.Pari method), 12 stirling() (cypari2.pari_instance.Pari_auto method), 348 Str() (cypari2.gen.Gen method), 384 Strchr() (cypari2.gen.Gen_base method), 428 strchr() (cvpari2.gen.Gen base method), 739 Strchr() (cypari2.pari_instance.Pari_auto method), 18 strchr() (cypari2.pari instance.Pari auto method), 349 Strexpand() (cypari2.gen.Gen method), 385 strjoin() (cypari2.gen.Gen_base method), 739 strjoin() (cypari2.pari instance.Pari auto method), 349 strsplit() (cypari2.gen.Gen_base method), 739 strsplit() (cypari2.pari_instance.Pari_auto method), 349 Strtex() (cypari2.gen.Gen method), 385 strtime() (cypari2.pari_instance.Pari_auto method), 349 subgrouplist() (cypari2.gen.Gen_base method), 740 (cypari2.pari_instance.Pari_auto subgrouplist() method), 349 subst() (cypari2.gen.Gen_base method), 740

subst() (cypari2.pari_instance.Pari_auto method), 350
substpol() (cypari2.gen.Gen base method), 741

345

substpol() (cypari2.pari_instance.Pari_auto method), 351 substvec() (cypari2.gen.Gen base method), 741 substvec() (cypari2.pari_instance.Pari_auto method), 351 sumdedekind() (cypari2.gen.Gen base method), 741 sumdedekind() (cypari2.pari instance.Pari auto method), 351 sumdigits() (cypari2.gen.Gen_base method), 741 sumdigits() (cypari2.pari_instance.Pari_auto method), 351 sumdiv() (cypari2.gen.Gen method), 422 sumdivk() (cypari2.gen.Gen method), 422 sumeulerrat() (cypari2.gen.Gen_base method), 741 sumeulerrat() (cypari2.pari_instance.Pari_auto method), 351 sumformal() (cypari2.gen.Gen_base method), 741 sumformal() (cypari2.pari_instance.Pari_auto method), 351 sumnumapinit() (cypari2.gen.Gen base method), 742 sumnumapinit() (cypari2.pari_instance.Pari_auto method), 352 sumnuminit() (cypari2.gen.Gen_base method), 742 sumnuminit() (cypari2.pari instance.Pari auto method), 352 sumnumlagrangeinit() (cypari2.gen.Gen_base method), 743 sumnumlagrangeinit() (*cy*method), pari2.pari_instance.Pari_auto 353 sumnummonieninit() (cypari2.gen.Gen_base method), 744 sumnummonieninit() (cypari2.pari_instance.Pari_auto method), 354 sumnumrat() (cypari2.gen.Gen base method), 745 sumnumrat() (cypari2.pari_instance.Pari_auto method), 355 system() (cypari2.pari_instance.Pari_auto method), 356 Т tan() (cypari2.gen.Gen base method), 746 tan() (cypari2.pari_instance.Pari_auto method), 356 tanh() (cypari2.gen.Gen_base method), 746 tanh() (cypari2.pari_instance.Pari_auto method), 356 taylor() (cypari2.gen.Gen_base method), 746 taylor() (cypari2.pari_instance.Pari_auto method),

- teichmuller() (cypari2.gen.Gen_base method), 746 teichmuller() (cypari2.pari_instance.Pari_auto
- method), 356 theta() (cypari2.gen.Gen_base method), 747

356

- theta() (cypari2.pari_instance.Pari_auto method), 357
- thetanullk() (cypari2.gen.Gen_base method), 747

- thetanullk() (cypari2.pari_instance.Pari_auto method), 357
- thue() (cypari2.gen.Gen base method), 747
- thue() (cypari2.pari_instance.Pari_auto method), 357
- thueinit() (cypari2.gen.Gen base method), 748
- thueinit() (cypari2.pari instance.Pari auto method), 358
- trace() (cypari2.gen.Gen_base method), 749
- trace() (cypari2.pari instance.Pari auto method), 359
- truncate() (cypari2.gen.Gen method), 423
- truncate() (cypari2.gen.Gen_base method), 749
- truncate() (cypari2.pari_instance.Pari_auto method), 359
- type() (cypari2.gen.Gen method), 423
- type() (cypari2.gen.Gen_base method), 749
- type() (cypari2.pari_instance.Pari_auto method), 359

U

unexportall() (cypari2.pari_instance.Pari_auto method), 359

V

- valuation() (cypari2.gen.Gen_base method), 749
- valuation() (cypari2.pari_instance.Pari_auto method), 359
- varhigher() (cypari2.pari_instance.Pari_auto method), 359
- variable() (cypari2.gen.Gen_base method), 750
- variable() (cypari2.pari_instance.Pari_auto method), 360
- variables() (cypari2.gen.Gen_base method), 750
- variables() (cypari2.pari_instance.Pari_auto method), 361
- varlower() (cypari2.pari_instance.Pari_auto method), 361
- Vec() (cypari2.gen.Gen base method), 429
- Vec() (cypari2.pari_instance.Pari_auto method), 18
- vecextract() (cypari2.gen.Gen base method), 750
- (cypari2.pari_instance.Pari_auto vecextract() method), 363
- vecmax() (cypari2.gen.Gen method), 424
- vecmax() (cypari2.gen.Gen_base method), 751
- vecmax() (cypari2.pari_instance.Pari_auto method), 363
- vecmin() (cypari2.gen.Gen method), 424
- vecmin() (cypari2.gen.Gen_base method), 752
- vecmin() (cypari2.pari_instance.Pari_auto method), 364
- vecprod() (cypari2.gen.Gen_base method), 752
- vecprod() (cypari2.pari_instance.Pari_auto method), 364
- Vecrev() (cypari2.gen.Gen_base method), 429
- Vecrev() (cypari2.pari instance.Pari auto method), 19
- vecsearch() (cypari2.gen.Gen_base method), 752

vecsearch() (cypari2.pari_instance.Pari_auto method), 364

- Vecsmall() (cypari2.gen.Gen_base method), 430
- Vecsmall() (cypari2.pari_instance.Pari_auto method), 19
- vecsort() (cypari2.gen.Gen_base method), 753
- vecsum() (cypari2.gen.Gen_base method), 754
- vecsum() (cypari2.pari_instance.Pari_auto method), 366
- vector() (cypari2.pari_instance.Pari method), 13
- version() (cypari2.pari_instance.Pari method), 13

W

weber() (cypari2.gen.Gen_base method), 754

- weber() (cypari2.pari_instance.Pari_auto method), 367

Ζ

- zero() (cypari2.pari_instance.Pari method), 13
- zeta() (cypari2.gen.Gen_base method), 754
- zeta() (cypari2.pari_instance.Pari_auto method), 367
- zetahurwitz() (cypari2.gen.Gen_base method), 755
- zetamult() (cypari2.gen.Gen_base method), 755
- zetamultall() (cypari2.pari_instance.Pari_auto method), 369

- zetamultdual() (cypari2.gen.Gen_base method), 756
- Zn_issquare() (cypari2.gen.Gen method), 386
- Zn_sqrt() (cypari2.gen.Gen method), 386
- znchar() (cypari2.gen.Gen_base method), 756
- zncharconductor() (cypari2.gen.Gen_base method),
 757
- znchardecompose() (cypari2.gen.Gen_base method),
 757

<pre>znchargauss() (cypari2.gen.Gen_base method), 758</pre>
znchargauss() (cypari2.pari_instance.Pari_auto
<i>method</i>), 372
<pre>zncharinduce() (cypari2.gen.Gen_base method), 758</pre>
<pre>zncharinduce() (cypari2.pari_instance.Pari_auto</pre>
method), 372
zncharisodd() (cypari2.gen.Gen base method), 759
zncharisodd() (cypari2.pari instance.Pari auto
method), 373
znchartokronecker() (cypari2.gen.Gen base
method), 759
znchartokronecker() (cv
pari2.pari instance.Pari auto method)
373
<pre>znchartoprimitive() (cypari2.gen.Gen_base</pre>
method), 759
znchartoprimitive() (cy
pari2.pari_instance.Pari_auto method) 374
znconreychar() (cypari2.gen.Gen base method), 760
znconreychar() (cypari2.pari instance.Pari auto
method). 374
znconrevconductor() (cvpari2.gen.Gen base
method). 761
znconrevconductor() (cv
pari2.pari instance.Pari auto method)
375
znconreyexp() (cypari2.gen.Gen base method), 762
znconreyexp() (cypari2.pari instance.Pari auto
method), 376
znconreylog() (cypari2.gen.Gen base method), 762
znconrevlog() (cvpari2.pari instance.Pari auto
method), 376
zncoppersmith() (cvpari2.gen.Gen base method), 763
zncoppersmith() (cvpari2.pari instance.Pari auto
method), 377
znlog() (cypari2.gen.Gen base method), 764
znlog() (cvpari2.pari instance.Pari auto method). 378
znorder() (cvpari2.gen.Gen base method), 766
znorder() (cvpari2.pari instance.Pari auto method)
380
<pre>znprimroot() (cvpari2.gen.Gen base method), 766</pre>
znprimroot() (cvpari2.pari instance.Pari auto
method), 380
znstar() (cypari2.gen.Gen base method). 766
<pre>znstar() (cypari2.pari_instance.Pari auto method)</pre>
<pre>znstar() (cypari2.pari_instance.Pari_auto method)</pre>