Sage Reference Manual: C/C++ Library Interfaces

Release 7.4

The Sage Development Team

Oct 20, 2016

CONTENTS

An underlying philosophy in the development of Sage is that it should provide unified library-level access to the some of the best GPL'd C/C++ libraries. Currently Sage provides some access to MWRANK, NTL, PARI, and Hanke, each of which are included with Sage.

The interfaces are implemented via shared libraries and data is moved between systems purely in memory. In particular, there is no interprocess interpreter parsing (e.g., `expect`), since everything is linked together and run as a single process. This is much more robust and efficient than using `expect`.

Each of these interfaces is used by other parts of Sage. For example, mwrank is used by the elliptic curves module to compute ranks of elliptic curves, and PARI is used for computation of class groups. It is thus probably not necessary for a casual user of Sage to be aware of the modules described in this chapter.

CHAPTER

ONE

SAGE INTERFACE TO CREMONA'S ECLIB LIBRARY (ALSO KNOWN AS MWRANK)

This is the Sage interface to John Cremona's `eclib` C++ library for arithmetic on elliptic curves. The classes defined in this module give Sage interpreter-level access to some of the functionality of `eclib`. For most purposes, it is not necessary to directly use these classes. Instead, one can create an `EllipticCurve` and call methods that are implemented using this module.

Note: This interface is a direct library-level interface to `eclib`, including the 2-descent program `mwrank`.

`sage.libs.eclib.interface.`**`get_precision`**`()`

Return the global NTL real number precision.

See also *`set_precision()`*.

Warning: The internal precision is binary. This function multiplies the binary precision by 0.3 ($=\log_2(10)$ approximately) and truncates.

OUTPUT:

(int) The current decimal precision.

EXAMPLES:

```
sage: mwrank_get_precision()
50
```

class `sage.libs.eclib.interface.`**`mwrank_EllipticCurve`**`(`*ainvs*, *verbose=False*`)`

Bases: `sage.structure.sage_object.SageObject`

The *`mwrank_EllipticCurve`* class represents an elliptic curve using the `Curvedata` class from `eclib`, called here an 'mwrank elliptic curve'.

Create the mwrank elliptic curve with invariants `ainvs`, which is a list of 5 or less *integers* a_1, a_2, a_3, a_4, and a_5.

If strictly less than 5 invariants are given, then the *first* ones are set to 0, so, e.g., `[3,4]` means $a_1 = a_2 = a_3 = 0$ and $a_4 = 3$, $a_5 = 4$.

INPUT:

- `ainvs` (list or tuple) – a list of 5 or less integers, the coefficients of a nonsingular Weierstrass equation.
- `verbose` (bool, default `False`) – verbosity flag. If `True`, then all Selmer group computations will be verbose.

EXAMPLES:

We create the elliptic curve $y^2 + y = x^3 + x^2 - 2x$:

```
sage: e = mwrank_EllipticCurve([0, 1, 1, -2, 0])
sage: e.ainvs()
[0, 1, 1, -2, 0]
```

This example illustrates that omitted a-invariants default to 0:

```
sage: e = mwrank_EllipticCurve([3, -4])
sage: e
y^2 = x^3 + 3*x - 4
sage: e.ainvs()
[0, 0, 0, 3, -4]
```

The entries of the input list are coerced to `int`. If this is impossible, then an error is raised:

```
sage: e = mwrank_EllipticCurve([3, -4.8]); e
Traceback (most recent call last):
...
TypeError: ainvs must be a list or tuple of integers.
```

When you enter a singular model you get an exception:

```
sage: e = mwrank_EllipticCurve([0, 0])
Traceback (most recent call last):
...
ArithmeticError: Invariants (= 0,0,0,0,0) do not describe an elliptic curve.
```

CPS_height_bound ( )

Return the Cremona-Prickett-Siksek height bound. This is a floating point number B such that if P is a point on the curve, then the naive logarithmic height $h(P)$ is less than $B + \hat{h}(P)$, where $\hat{h}(P)$ is the canonical height of P.

> **Warning:** We assume the model is minimal!

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.CPS_height_bound()
14.163198527061496
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.CPS_height_bound()
0.0
```

ainvs ( )

Returns the a-invariants of this mwrank elliptic curve.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-1,0])
sage: E.ainvs()
[0, 0, 1, -1, 0]
```

certain ( )

Returns `True` if the last *`two_descent()`* call provably correctly computed the rank. If *two_descent()* hasn't been called, then it is first called by *certain()* using the default parameters.

The result is `True` if and only if the results of the methods *rank()* and *rank_bound()* are equal.

EXAMPLES:

A 2-descent does not determine $E(\mathbf{Q})$ with certainty for the curve $y^2 + y = x^3 - x^2 - 120x - 2183$:

```
sage: E = mwrank_EllipticCurve([0, -1, 1, -120, -2183])
sage: E.two_descent(False)
...
sage: E.certain()
False
sage: E.rank()
0
```

The previous value is only a lower bound; the upper bound is greater:

```
sage: E.rank_bound()
2
```

In fact the rank of E is actually 0 (as one could see by computing the L-function), but Sha has order 4 and the 2-torsion is trivial, so mwrank cannot conclusively determine the rank in this case.

conductor ()

Return the conductor of this curve, computed using Cremona's implementation of Tate's algorithm.

Note: This is independent of PARI's.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([1, 1, 0, -6958, -224588])
sage: E.conductor()
2310
```

gens ()

Return a list of the generators for the Mordell-Weil group.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.gens()
[[0, -1, 1]]
```

isogeny_class (*verbose=False*)

Returns the isogeny class of this mwrank elliptic curve.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.isogeny_class()
([[0, -1, 1, 0, 0], [0, -1, 1, -10, -20], [0, -1, 1, -7820, -263580]], [[0,
↪5, 0], [5, 0, 5], [0, 5, 0]])
```

rank ()

Returns the rank of this curve, computed using *two_descent()*.

In general this may only be a lower bound for the rank; an upper bound may be obtained using the function `rank_bound()` . To test whether the value has been proved to be correct, use the method `certain()` .

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.rank()
0
sage: E.certain()
True
```

```
sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank()
0
sage: E.certain()
False
```

rank_bound ( )

Returns an upper bound for the rank of this curve, computed using `two_descent()` .

If the curve has no 2-torsion, this is equal to the 2-Selmer rank. If the curve has 2-torsion, the upper bound may be smaller than the bound obtained from the 2-Selmer rank minus the 2-rank of the torsion, since more information is gained from the 2-isogenous curve or curves.

EXAMPLES:

The following is the curve 960D1, which has rank 0, but Sha of order 4:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.rank_bound()
0
sage: E.rank()
0
```

In this case the rank was computed using a second descent, which is able to determine (by considering a 2-isogenous curve) that Sha is nontrivial. If we deliberately stop the second descent, the rank bound is larger:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.two_descent(second_descent = False, verbose=False)
sage: E.rank_bound()
2
```

In contrast, for the curve 571A, also with rank 0 and Sha of order 4, we only obtain an upper bound of 2:

```
sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank_bound()
2
```

In this case the value returned by `rank()` is only a lower bound in general (though this is correct):

```
sage: E.rank()
0
sage: E.certain()
False
```

regulator ( )

Return the regulator of the saturated Mordell-Weil group.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.regulator()
0.05111140823996884
```

saturate (*bound=-1*)

Compute the saturation of the Mordell-Weil group at all primes up to `bound` .

INPUT:

- `bound` (int, default -1) – Use -1 (the default) to saturate at *all* primes, 0 for no saturation, or n (a positive integer) to saturate at all primes up to n.

EXAMPLES:

Since the 2-descent automatically saturates at primes up to 20, it is not easy to come up with an example where saturation has any effect:

```
sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.gens()
[[-1001107, -4004428, 1]]
sage: E.saturate()
sage: E.gens()
[[-1001107, -4004428, 1]]
```

Check that trac ticket #18031 is fixed:

```
sage: E = EllipticCurve([0,-1,1,-266,968])
sage: Q1 = E([-1995,3674,125])
sage: Q2 = E([157,1950,1])
sage: E.saturation([Q1,Q2])
([(1 : -27 : 1), (157 : 1950 : 1)], 3, 0.801588644684981)
```

selmer_rank ( )

Returns the rank of the 2-Selmer group of the curve.

EXAMPLES:

The following is the curve 960D1, which has rank 0, but Sha of order 4. The 2-torsion has rank 2, and the Selmer rank is 3:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.selmer_rank()
3
```

Nevertheless, we can obtain a tight upper bound on the rank since a second descent is performed which establishes the 2-rank of Sha:

```
sage: E.rank_bound()
0
```

To show that this was resolved using a second descent, we do the computation again but turn off `second_descent` :

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.two_descent(second_descent = False, verbose=False)
sage: E.rank_bound()
2
```

For the curve 571A, also with rank 0 and Sha of order 4, but with no 2-torsion, the Selmer rank is strictly greater than the rank:

```
sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.selmer_rank()
2
sage: E.rank_bound()
2
```

In cases like this with no 2-torsion, the rank upper bound is always equal to the 2-Selmer rank. If we ask for the rank, all we get is a lower bound:

```
sage: E.rank()
0
sage: E.certain()
False
```

set_verbose (*verbose*)

Set the verbosity of printing of output by the *two_descent()* and other functions.

INPUT:

- `verbose` (int) – if positive, print lots of output when doing 2-descent.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.saturate() # no output
sage: E.gens()
[[0, -1, 1]]

sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.set_verbose(1)
sage: E.saturate() # tol 1e-14
Basic pair: I=48, J=-432
disc=255744
2-adic index bound = 2
By Lemma 5.1(a), 2-adic index = 1
2-adic index = 1
One (I,J) pair
Looking for quartics with I = 48, J = -432
Looking for Type 2 quartics:
Trying positive a from 1 up to 1 (square a first...)
(1,0,-6,4,1)        --trivial
Trying positive a from 1 up to 1 (...then non-square a)
Finished looking for Type 2 quartics.
Looking for Type 1 quartics:
Trying positive a from 1 up to 2 (square a first...)
(1,0,0,4,4) --nontrivial...(x:y:z) = (1 : 1 : 0)
Point = [0:0:1]
    height = 0.0511114082399688402358
Rank of B=im(eps) increases to 1 (The previous point is on the egg)
Exiting search for Type 1 quartics after finding one which is globally
↪soluble.
Mordell rank contribution from B=im(eps) = 1
Selmer  rank contribution from B=im(eps) = 1
Sha     rank contribution from B=im(eps) = 0
Mordell rank contribution from A=ker(eps) = 0
Selmer  rank contribution from A=ker(eps) = 0
```

```
Sha     rank contribution from A=ker(eps) = 0
Searching for points (bound = 8)...done:
  found points which generate a subgroup of rank 1
  and regulator 0.0511114082399688402358
Processing points found during 2-descent...done:
  now regulator = 0.0511114082399688402358
Saturating (with bound = -1)...done:
  points were already saturated.
```

silverman_bound ( )

Return the Silverman height bound. This is a floating point number B such that if P is a point on the curve, then the naive logarithmic height $h(P)$ is less than $B + \hat{h}(P)$, where $\hat{h}(P)$ is the canonical height of P.

> **Warning:** We assume the model is minimal!

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.silverman_bound()
18.29545210468247
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.silverman_bound()
6.284833369972403
```

two_descent (*verbose=True*, *selmer_only=False*, *first_limit=20*, *second_limit=8*, *n_aux=-1*, *second_descent=True*)

Compute 2-descent data for this curve.

INPUT:

- `verbose` (bool, default `True`) – print what mwrank is doing.
- `selmer_only` (bool, default `False`) – `selmer_only` switch.
- `first_limit` (int, default 20) – bound on $|x| + |z|$ in quartic point search.
- `second_limit` (int, default 8) – bound on $\log \max(|x|, |z|)$, i.e. logarithmic.
- `n_aux` (int, default -1) – (only relevant for general 2-descent when 2-torsion trivial) number of primes used for quartic search. `n_aux=-1` causes default (8) to be used. Increase for curves of higher rank.
- `second_descent` (bool, default `True`) – (only relevant for curves with 2-torsion, where mwrank uses descent via 2-isogeny) flag determining whether or not to do second descent. *Default strongly recommended.*

OUTPUT:

Nothing – nothing is returned.

TESTS:

See trac ticket #7992:

```
sage: EllipticCurve([0, prod(prime_range(10))]).mwrank_curve().two_descent()
Basic pair: I=0, J=-5670
disc=-32148900
2-adic index bound = 2
```

```
2-adic index = 2
Two (I,J) pairs
Looking for quartics with I = 0, J = -5670
Looking for Type 3 quartics:
Trying positive a from 1 up to 5 (square a first...)
Trying positive a from 1 up to 5 (...then non-square a)
(2,0,-12,19,-6)      --nontrivial...(x:y:z) = (2 : 4 : 1)
Point = [-2488:-4997:512]
    height = 6.46767239...
Rank of B=im(eps) increases to 1
Trying negative a from -1 down to -3
Finished looking for Type 3 quartics.
Looking for quartics with I = 0, J = -362880
Looking for Type 3 quartics:
Trying positive a from 1 up to 20 (square a first...)
Trying positive a from 1 up to 20 (...then non-square a)
Trying negative a from -1 down to -13
Finished looking for Type 3 quartics.
Mordell rank contribution from B=im(eps) = 1
Selmer  rank contribution from B=im(eps) = 1
Sha     rank contribution from B=im(eps) = 0
Mordell rank contribution from A=ker(eps) = 0
Selmer  rank contribution from A=ker(eps) = 0
Sha     rank contribution from A=ker(eps) = 0
sage: EllipticCurve([0, prod(prime_range(100))]).mwrank_curve().two_descent()
Traceback (most recent call last):
...
RuntimeError: Aborted
```

Calling this method twice does not cause a segmentation fault (see trac ticket #10665):

```
sage: E = EllipticCurve([1, 1, 0, 0, 528])
sage: E.two_descent(verbose=False)
True
sage: E.two_descent(verbose=False)
True
```

class `sage.libs.eclib.interface.`**`mwrank_MordellWeil`**(*curve*, *verbose=True*, *pp=1*, *maxr=999*)

Bases: `sage.structure.sage_object.SageObject`

The *`mwrank_MordellWeil`* class represents a subgroup of a Mordell-Weil group. Use this class to saturate a specified list of points on an *`mwrank_EllipticCurve`*, or to search for points up to some bound.

INPUT:

- `curve` (*`mwrank_EllipticCurve`*) – the underlying elliptic curve.
- `verbose` (bool, default `False`) – verbosity flag (controls amount of output produced in point searches).
- `pp` (int, default 1) – process points flag (if nonzero, the points found are processed, so that at all times only a $\mathbf{Z}$-basis for the subgroup generated by the points found so far is stored; if zero, no processing is done and all points found are stored).
- `maxr` (int, default 999) – maximum rank (quit point searching once the points found generate a subgroup of this rank; useful if an upper bound for the rank is already known).

EXAMPLE:

```
sage: E = mwrank_EllipticCurve([1,0,1,4,-6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ
Subgroup of Mordell-Weil group: []
sage: EQ.search(2)
P1 = [0:1:0]     is torsion point, order 1
P1 = [1:-1:1]    is torsion point, order 2
P1 = [2:2:1]     is torsion point, order 3
P1 = [9:23:1]    is torsion point, order 6

sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(2)
P1 = [0:1:0]     is torsion point, order 1
P1 = [-3:0:1]     is generator number 1
...
P4 = [-91:804:343]       = -2*P1 + 2*P2 + 1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

Example to illustrate the verbose parameter:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E, verbose=False)
sage: EQ.search(1)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]

sage: EQ = mwrank_MordellWeil(E, verbose=True)
sage: EQ.search(1)
P1 = [0:1:0]     is torsion point, order 1
P1 = [-3:0:1]     is generator number 1
saturating up to 20...Checking 2-saturation
Points have successfully been 2-saturated (max q used = 7)
Checking 3-saturation
Points have successfully been 3-saturated (max q used = 7)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 23)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 41)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 17)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 43)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 31)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 37)
done
P2 = [-2:3:1]     is generator number 2
saturating up to 20...Checking 2-saturation
possible kernel vector = [1,1]
This point may be in 2E(Q): [14:-52:1]
...and it is!
Replacing old generator #1 with new generator [1:-1:1]
Points have successfully been 2-saturated (max q used = 7)
Index gain = 2^1
Checking 3-saturation
```

```
Points have successfully been 3-saturated (max q used = 13)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 67)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 53)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 73)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 103)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 113)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 47)
done (index = 2).
Gained index 2, new generators = [ [1:-1:1] [-2:3:1] ]
P3 = [-14:25:8]   is generator number 3
saturating up to 20...Checking 2-saturation
Points have successfully been 2-saturated (max q used = 11)
Checking 3-saturation
Points have successfully been 3-saturated (max q used = 13)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 71)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 101)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 127)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 151)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 139)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 179)
done (index = 1).
P4 = [-1:3:1]    = -1*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [0:2:1]     = 2*P1 + 0*P2 + 1*P3 (mod torsion)
P4 = [2:13:8]    = -3*P1 + 1*P2 + -1*P3 (mod torsion)
P4 = [1:0:1]     = -1*P1 + 0*P2 + 0*P3 (mod torsion)
P4 = [2:0:1]     = -1*P1 + 1*P2 + 0*P3 (mod torsion)
P4 = [18:7:8]    = -2*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [3:3:1]     = 1*P1 + 0*P2 + 1*P3 (mod torsion)
P4 = [4:6:1]     = 0*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [36:69:64]  = 1*P1 + -2*P2 + 0*P3 (mod torsion)
P4 = [68:-25:64]         = -2*P1 + -1*P2 + -2*P3 (mod torsion)
P4 = [12:35:27]  = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

Example to illustrate the process points (pp) parameter:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E, verbose=False, pp=1)
sage: EQ.search(1); EQ # generators only
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
sage: EQ = mwrank_MordellWeil(E, verbose=False, pp=0)
sage: EQ.search(1); EQ # all points found
Subgroup of Mordell-Weil group: [[-3:0:1], [-2:3:1], [-14:25:8], [-1:3:1], [0:2:
↪1], [2:13:8], [1:0:1], [2:0:1], [18:7:8], [3:3:1], [4:6:1], [36:69:64], [68:-25:
↪64], [12:35:27]]
```

points ( )

Return a list of the generating points in this Mordell-Weil group.

OUTPUT:

(list) A list of lists of length 3, each holding the primitive integer coordinates $[x, y, z]$ of a generating point.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(1)
P1 = [0:1:0]         is torsion point, order 1
P1 = [-3:0:1]         is generator number 1
...
P4 = [12:35:27]      = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ.points()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
```

process (*v*, *sat=0*)

This function allows one to add points to a *mwrank_MordellWeil* object.

Process points in the list `v` , with saturation at primes up to `sat` . If `sat` is zero (the default), do no saturation.

INPUT:

- `v` (list of 3-tuples or lists of ints or Integers) – a list of triples of integers, which define points on the curve.
- `sat` (int, default 0) – saturate at primes up to `sat` , or at *all* primes if `sat` is zero.

OUTPUT:

None. But note that if the `verbose` flag is set, then there will be some output as a side-effect.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.gens()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1, -1, 1], [-2, 3, 1], [-14, 25, 8]])
P1 = [1:-1:1]         is generator number 1
P2 = [-2:3:1]         is generator number 2
P3 = [-14:25:8]       is generator number 3
```

```
sage: EQ.points()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
```

Example to illustrate the saturation parameter `sat` :

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191,␣
↪2969715140223272], [-13422227300, -49322830557, 12167000000]], sat=20)
P1 = [1547:-2967:343]         is generator number 1
...
Gained index 5, new generators = [ [-2:3:1] [-14:25:8] [1:-1:1] ]

sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [1, -1, 1]]
```

Here the processing was followed by saturation at primes up to 20. Now we prevent this initial saturation:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191,␣
↪2969715140223272], [-13422227300, -49322830557, 12167000000]], sat=0)
P1 = [1547:-2967:343]         is generator number 1
P2 = [2707496766203306:864581029138191:2969715140223272]      is generator␣
↪number 2
P3 = [-13422227300:-49322830557:12167000000]          is generator number 3
sage: EQ.points()
[[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272], [-
↪13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
375.42919921875
sage: EQ.saturate(2)  # points were not 2-saturated
saturating basis...Saturation index bound = 93
WARNING: saturation at primes p > 2 will not be done;
...
Gained index 2
New regulator =  93.857300720636393209
(False, 2, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [2707496766203306, 864581029138191, 2969715140223272], [-
↪13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
93.8572998046875
sage: EQ.saturate(3)  # points were not 3-saturated
saturating basis...Saturation index bound = 46
WARNING: saturation at primes p > 3 will not be done;
...
Gained index 3
New regulator =  10.4285889689595992455
(False, 3, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
10.4285888671875
sage: EQ.saturate(5)  # points were not 5-saturated
saturating basis...Saturation index bound = 15
WARNING: saturation at primes p > 5 will not be done;
...
Gained index 5
New regulator =  0.417143558758383969818
(False, 5, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [1, -1, 1]]
sage: EQ.regulator()
0.4171435534954071
sage: EQ.saturate()   # points are now saturated
saturating basis...Saturation index bound = 3
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
```

```
(True, 1, '[ ]')
```

rank ( )

Return the rank of this subgroup of the Mordell-Weil group.

OUTPUT:

(int) The rank of this subgroup of the Mordell-Weil group.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.rank()
0
```

A rank 3 example:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.rank()
0
sage: EQ.regulator()
1.0
```

The preceding output is correct, since we have not yet tried to find any points on the curve either by searching or 2-descent:

```
sage: EQ
Subgroup of Mordell-Weil group: []
```

Now we do a very small search:

```
sage: EQ.search(1)
P1 = [0:1:0]         is torsion point, order 1
P1 = [-3:0:1]         is generator number 1
saturating up to 20...Checking 2-saturation
...
P4 = [12:35:27]       = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
sage: EQ.rank()
3
sage: EQ.regulator()
0.4171435534954071
```

We do in fact now have a full Mordell-Weil basis.

regulator ( )

Return the regulator of the points in this subgroup of the Mordell-Weil group.

Note: `eclib` can compute the regulator to arbitrary precision, but the interface currently returns the output as a `float`.

OUTPUT:

(float) The regulator of the points in this subgroup.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.regulator()
1.0

sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.regulator()
0.417143558758384
```

saturate (*max_prime=-1*, *odd_primes_only=False*)

Saturate this subgroup of the Mordell-Weil group.

INPUT:

- `max_prime` (int, default -1) – saturation is performed for all primes up to `max_prime`. If -1 (the default), an upper bound is computed for the primes at which the subgroup may not be saturated, and this is used; however, if the computed bound is greater than a value set by the `eclib` library (currently 97) then no saturation will be attempted at primes above this.
- `odd_primes_only` (bool, default `False`) – only do saturation at odd primes. (If the points have been found via :meth:`two_descent()` they should already be 2-saturated.)

OUTPUT:

(3-tuple) (`ok`, `index`, `unsatlist`) where:

- `ok` (bool) – `True` if and only if the saturation was provably successful at all primes attempted. If the default was used for `max_prime` and no warning was output about the computed saturation bound being too high, then `True` indicates that the subgroup is saturated at *all* primes.
- `index` (int) – the index of the group generated by the original points in their saturation.
- `unsatlist` (list of ints) – list of primes at which saturation could not be proved or achieved. Increasing the decimal precision should correct this, since it happens when a linear combination of the points appears to be a multiple of p but cannot be divided by p. (Note that `eclib` uses floating point methods based on elliptic logarithms to divide points.)

Note: We emphasize that if this function returns `True` as the first return argument (`ok`), and if the default was used for the parameter `max_prime`, then the points in the basis after calling this function are saturated at *all* primes, i.e., saturating at the primes up to `max_prime` are sufficient to saturate at all primes. Note that the function might not have needed to saturate at all primes up to `max_prime`. It has worked out what prime you need to saturate up to, and that prime might be smaller than `max_prime`.

Note: Currently (May 2010), this does not remember the result of calling *`search()`*. So calling *`search()`* up to height 20 then calling *`saturate()`* results in another search up to height 18.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
```

We initialise with three points which happen to be 2, 3 and 5 times the generators of this rank 3 curve. To prevent automatic saturation at this stage we set the parameter `sat` to 0 (which is in fact the default):

```
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191,
↪2969715140223272], [-13422227300, -49322830557, 12167000000]], sat=0)
P1 = [1547:-2967:343]          is generator number 1
```

```
P2 = [2707496766203306:864581029138191:2969715140223272]      is generator␣
↪number 2
P3 = [-13422227300:-49322830557:12167000000]          is generator number 3
sage: EQ
Subgroup of Mordell-Weil group: [[1547:-2967:343], [2707496766203306:
↪864581029138191:2969715140223272], [-13422227300:-49322830557:12167000000]]
sage: EQ.regulator()
375.42919921875
```

Now we saturate at $p = 2$, and gain index 2:

```
sage: EQ.saturate(2)  # points were not 2-saturated
saturating basis...Saturation index bound = 93
WARNING: saturation at primes p > 2 will not be done;
...
Gained index 2
New regulator =  93.857300720636393209
(False, 2, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [2707496766203306:864581029138191:
↪2969715140223272], [-13422227300:-49322830557:12167000000]]
sage: EQ.regulator()
93.8572998046875
```

Now we saturate at $p = 3$, and gain index 3:

```
sage: EQ.saturate(3)  # points were not 3-saturated
saturating basis...Saturation index bound = 46
WARNING: saturation at primes p > 3 will not be done;
...
Gained index 3
New regulator =  10.4285889689595992455
(False, 3, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [-13422227300:-
↪49322830557:12167000000]]
sage: EQ.regulator()
10.4285888671875
```

Now we saturate at $p = 5$, and gain index 5:

```
sage: EQ.saturate(5)  # points were not 5-saturated
saturating basis...Saturation index bound = 15
WARNING: saturation at primes p > 5 will not be done;
...
Gained index 5
New regulator =  0.417143558758383969818
(False, 5, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.4171435534954071
```

Finally we finish the saturation. The output here shows that the points are now provably saturated at all primes:

```
sage: EQ.saturate()   # points are now saturated
saturating basis...Saturation index bound = 3
```

```
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
(True, 1, '[ ]')
```

Of course, the `process()` function would have done all this automatically for us:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191,␣
↪2969715140223272], [-13422227300, -49322830557, 12167000000]], sat=5)
P1 = [1547:-2967:343]         is generator number 1
...
Gained index 5, new generators = [ [-2:3:1] [-14:25:8] [1:-1:1] ]
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.4171435534954071
```

But we would still need to use the `saturate()` function to verify that full saturation has been done:

```
sage: EQ.saturate()
saturating basis...Saturation index bound = 3
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
(True, 1, '[ ]')
```

Note the output of the preceding command: it proves that the index of the points in their saturation is at most 3, then proves saturation at 2 and at 3, by reducing the points modulo all primes of good reduction up to 11, respectively 13.

search (*height_limit=18*, *verbose=False*)

Search for new points, and add them to this subgroup of the Mordell-Weil group.

INPUT:

- `height_limit` (float, default: 18) – search up to this logarithmic height.

Note: On 32-bit machines, this *must* be < 21.48 else $\exp(h_{\text{lim}}) > 2^{31}$ and overflows. On 64-bit machines, it must be *at most* 43.668. However, this bound is a logarithmic bound and increasing it by just 1 increases the running time by (roughly) $\exp(1.5) = 4.5$, so searching up to even 20 takes a very long time.

Note: The search is carried out with a quadratic sieve, using code adapted from a version of Michael Stoll's `ratpoints` program. It would be preferable to use a newer version of `ratpoints`.

- `verbose` (bool, default `False`) – turn verbose operation on or off.

EXAMPLES:

A rank 3 example, where a very small search is sufficient to find a Mordell-Weil basis:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(1)
P1 = [0:1:0]         is torsion point, order 1
P1 = [-3:0:1]         is generator number 1
...
P4 = [12:35:27]       = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

In the next example, a search bound of 12 is needed to find a non-torsion point:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -18392, -1186248]) #1056g4
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(11); EQ
P1 = [0:1:0]         is torsion point, order 1
P1 = [161:0:1]       is torsion point, order 2
Subgroup of Mordell-Weil group: []
sage: EQ.search(12); EQ
P1 = [0:1:0]         is torsion point, order 1
P1 = [161:0:1]       is torsion point, order 2
P1 = [4413270:10381877:27000]        is generator number 1
...
Subgroup of Mordell-Weil group: [[4413270:10381877:27000]]
```

sage.libs.eclib.interface.**set_precision** (*n*)

Set the global NTL real number precision. This has a massive effect on the speed of mwrank calculations. The default (used if this function is not called) is `n=50`, but it might have to be increased if a computation fails. See also *get_precision()*.

INPUT:

- `n` (long) – real precision used for floating point computations in the library, in decimal digits.

Warning: This change is global and affects *all* future calls of eclib functions by Sage.

EXAMPLES:

```
sage: mwrank_set_precision(20)
```

CHAPTER

TWO

CYTHON INTERFACE TO CREMONA'S ECLIB LIBRARY (ALSO KNOWN AS MWRANK)

EXAMPLES:

```
sage: from sage.libs.eclib.mwrank import _Curvedata, _mw
sage: c = _Curvedata(1,2,3,4,5)

sage: print(c)
[1,2,3,4,5]
b2 = 9        b4 = 11          b6 = 29          b8 = 35
c4 = -183            c6 = -3429
disc = -10351        (# real components = 1)
#torsion not yet computed

sage: t= _mw(c)
sage: t.search(10)
sage: t
[[1:2:1]]
```

sage.libs.eclib.mwrank.**get_precision**()

Returns the working floating point precision of mwrank.

OUTPUT:

(int) The current precision in decimal digits.

EXAMPLE:

```
sage: from sage.libs.eclib.mwrank import get_precision
sage: get_precision()
50
```

sage.libs.eclib.mwrank.**initprimes**(*filename*, *verb=False*)

Initialises mwrank/eclib's internal prime list.

INPUT:

- `filename` (string) – the name of a file of primes.
- `verb` (bool: default `False`) – verbose or not?

EXAMPLES:

```
sage: file = os.path.join(SAGE_TMP, 'PRIMES')
sage: open(file,'w').write(' '.join([str(p) for p in prime_range(10^7,10^7+20)]))
sage: mwrank_initprimes(file, verb=True)
Computed 78519 primes, largest is 1000253
```

```
reading primes from file ...
read extra prime 10000019
finished reading primes from file ...
Extra primes in list: 10000019

sage: mwrank_initprimes("x" + file, True)
Traceback (most recent call last):
...
IOError: No such file or directory: ...
```

sage.libs.eclib.mwrank.**set_precision**(*n*)

Sets the working floating point precision of mwrank.

INPUT:

- n (int) – a positive integer: the number of decimal digits.

OUTPUT:

None.

EXAMPLE:

```
sage: from sage.libs.eclib.mwrank import set_precision
sage: set_precision(50)
```

CHAPTER

THREE

CREMONA MATRICES

class `sage.libs.eclib.mat.`**`Matrix`**

Bases: `object`

A Cremona Matrix.

EXAMPLES:

```
sage: M = CremonaModularSymbols(225)
sage: t = M.hecke_matrix(2)
sage: type(t)
<type 'sage.libs.eclib.mat.Matrix'>
sage: t
61 x 61 Cremona matrix over Rational Field
```

TESTS:

```
sage: t = CremonaModularSymbols(11).hecke_matrix(2); t
3 x 3 Cremona matrix over Rational Field
sage: type(t)
<type 'sage.libs.eclib.mat.Matrix'>
```

`add_scalar` (*s*)

Return new matrix obtained by adding s to each diagonal entry of self.

EXAMPLES:

```
sage: M = CremonaModularSymbols(23, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2); print(t.str())
[ 0  1]
[ 1 -1]
sage: w = t.add_scalar(3); print(w.str())
[3 1]
[1 2]
```

`charpoly` (*var='x'*)

Return the characteristic polynomial of this matrix, viewed as as a matrix over the integers.

ALGORITHM:

Note that currently, this function converts this matrix into a dense matrix over the integers, then calls the charpoly algorithm on that, which I think is LinBox's.

EXAMPLES:

```
sage: M = CremonaModularSymbols(33, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2)
```

```
sage: t.charpoly()
x^3 + 3*x^2 - 4
sage: t.charpoly().factor()
(x - 1) * (x + 2)^2
```

ncols ( )

Return the number of columns of this matrix.

EXAMPLES:

```
sage: M = CremonaModularSymbols(1234, sign=1)
sage: t = M.hecke_matrix(3); t.ncols()
156
sage: M.dimension()
156
```

nrows ( )

Return the number of rows of this matrix.

EXAMPLES:

```
sage: M = CremonaModularSymbols(19, sign=1)
sage: t = M.hecke_matrix(13); t
2 x 2 Cremona matrix over Rational Field
sage: t.nrows()
2
```

sage_matrix_over_ZZ (*sparse=True*)

Return corresponding Sage matrix over the integers.

INPUT:

- `sparse` – (default: True) whether the return matrix has a sparse representation

EXAMPLES:

```
sage: M = CremonaModularSymbols(23, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2)
sage: s = t.sage_matrix_over_ZZ(); s
[ 0  1]
[ 1 -1]
sage: type(s)
<type 'sage.matrix.matrix_integer_sparse.Matrix_integer_sparse'>
sage: s = t.sage_matrix_over_ZZ(sparse=False); s
[ 0  1]
[ 1 -1]
sage: type(s)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
```

str ( )

Return full string representation of this matrix, never in compact form.

EXAMPLES:

```
sage: M = CremonaModularSymbols(22, sign=1)
sage: t = M.hecke_matrix(13)
sage: t.str()
'[14  0  0  0  0]\n[-4 12  0  8  4]\n[ 0 -6  4 -6  0]\n[ 4  2  0  6 -4]\n[ 0 
↪0  0  0 14]'
```

class `sage.libs.eclib.mat.`**`MatrixFactory`**

Bases: `object`

CHAPTER

FOUR

MODULAR SYMBOLS USING ECLIB NEWFORMS

class `sage.libs.eclib.newforms.`**`ECModularSymbol`**

Bases: `object`

Modular symbol associated with an elliptic curve, using John Cremona's newforms class.

EXAMPLES:

```
sage: from sage.libs.eclib.newforms import ECModularSymbol
sage: E = EllipticCurve('11a')
sage: M = ECModularSymbol(E,1); M
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined␣
↪by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [M(1/i) for i in range(1,11)]
[0, 2, 1, -1, -2, -2, -1, 1, 2, 0]
```

The curve is automatically converted to its minimal model:

```
sage: E = EllipticCurve([0,0,0,0,1/4])
sage: ECModularSymbol(E)
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined␣
↪by y^2 + y = x^3 over Rational Field
```

CHAPTER

FIVE

CREMONA MODULAR SYMBOLS

class sage.libs.eclib.homspace.**ModularSymbols**

Bases: object

Class of Cremona Modular Symbols of given level and sign (and weight 2).

EXAMPLES:

```
sage: M = CremonaModularSymbols(225)
sage: type(M)
<type 'sage.libs.eclib.homspace.ModularSymbols'>
```

dimension ( )

Return the dimension of this modular symbols space.

EXAMPLES:

```
sage: M = CremonaModularSymbols(1234, sign=1)
sage: M.dimension()
156
```

hecke_matrix (*p*, *dual=False*, *verbose=False*)

Return the matrix of the p -th Hecke operator acting on this space of modular symbols.

The result of this command is not cached.

INPUT:

- p – a prime number
- **dual – (default: False) whether to compute the Hecke** operator acting on the dual space, i.e., the transpose of the Hecke operator
- verbose – (default: False) print verbose output

OUTPUT:

(matrix) If p divides the level, the matrix of the Atkin-Lehner involution W_p at p ; otherwise the matrix of the Hecke operator T_p,

EXAMPLES:

```
sage: M = CremonaModularSymbols(37)
sage: t = M.hecke_matrix(2); t
5 x 5 Cremona matrix over Rational Field
sage: print(t.str())
[ 3  0  0  0  0]
[-1 -1  1  1  0]
[ 0  0 -1  0  1]
```

```
[-1  1  0 -1 -1]
[ 0  0  1  0 -1]
sage: t.charpoly().factor()
(x - 3) * x^2 * (x + 2)^2
sage: print(M.hecke_matrix(2, dual=True).str())
[ 3 -1  0 -1  0]
[ 0 -1  0  1  0]
[ 0  1 -1  0  1]
[ 0  1  0 -1  0]
[ 0  0  1 -1 -1]
sage: w = M.hecke_matrix(37); w
5 x 5 Cremona matrix over Rational Field
sage: w.charpoly().factor()
(x - 1)^2 * (x + 1)^3
sage: sw = w.sage_matrix_over_ZZ()
sage: st = t.sage_matrix_over_ZZ()
sage: sw^2 == sw.parent()(1)
True
sage: st*sw == sw*st
True
```

is_cuspidal ( )

Return whether or not this space is cuspidal.

EXAMPLES:

```
sage: M = CremonaModularSymbols(1122); M.is_cuspidal()
0
sage: M = CremonaModularSymbols(1122, cuspidal=True); M.is_cuspidal()
1
```

level ( )

Return the level of this modular symbols space.

EXAMPLES:

```
sage: M = CremonaModularSymbols(1234, sign=1)
sage: M.level()
1234
```

number_of_cusps ( )

Return the number of cusps for $\Gamma_0(N)$, where N is the level.

EXAMPLES:

```
sage: M = CremonaModularSymbols(225)
sage: M.number_of_cusps()
24
```

sign ( )

Return the sign of this Cremona modular symbols space. The sign is either 0, +1 or -1.

EXAMPLES:

```
sage: M = CremonaModularSymbols(1122, sign=1); M
Cremona Modular Symbols space of dimension 224 for Gamma_0(1122) of weight 2␣
↪with sign 1
sage: M.sign()
1
```

```
sage: M = CremonaModularSymbols(1122); M
Cremona Modular Symbols space of dimension 433 for Gamma_0(1122) of weight 2␣
↪with sign 0
sage: M.sign()
0
sage: M = CremonaModularSymbols(1122, sign=-1); M
Cremona Modular Symbols space of dimension 209 for Gamma_0(1122) of weight 2␣
↪with sign -1
sage: M.sign()
-1
```

CHAPTER

SIX

CREMONA MODULAR SYMBOLS

sage.libs.eclib.constructor.**CremonaModularSymbols**(*level*, *sign=0*, *cuspidal=False*, *verbose=0*)

Return the space of Cremona modular symbols with given level, sign, etc.

INPUT:

- `level` – an integer >= 2 (at least 2, not just positive!)
- `sign` – an integer either 0 (the default) or 1 or -1.
- `cuspidal` – (default: False); if True, compute only the cuspidal subspace
- `verbose` – (default: False): if True, print verbose information while creating space

EXAMPLES:

```
sage: M = CremonaModularSymbols(43); M
Cremona Modular Symbols space of dimension 7 for Gamma_0(43) of weight 2 with␣
↪sign 0
sage: M = CremonaModularSymbols(43, sign=1); M
Cremona Modular Symbols space of dimension 4 for Gamma_0(43) of weight 2 with␣
↪sign 1
sage: M = CremonaModularSymbols(43, cuspidal=True); M
Cremona Cuspidal Modular Symbols space of dimension 6 for Gamma_0(43) of weight 2␣
↪with sign 0
sage: M = CremonaModularSymbols(43, cuspidal=True, sign=1); M
Cremona Cuspidal Modular Symbols space of dimension 3 for Gamma_0(43) of weight 2␣
↪with sign 1
```

When run interactively, the following command will display verbose output:

```
sage: M = CremonaModularSymbols(43, verbose=1)
After 2-term relations, ngens = 22
ngens     = 22
maxnumrel = 32
relation matrix has = 704 entries...
Finished 3-term relations: numrel = 16 ( maxnumrel = 32)
relmat has 42 nonzero entries (density = 0.0596591)
Computing kernel...
time to compute kernel =  (... seconds)
rk = 7
Number of cusps is 2
ncusps = 2
About to compute matrix of delta
delta matrix done: size 2x7.
About to compute kernel of delta
done
```

```
Finished constructing homspace.
sage: M
Cremona Modular Symbols space of dimension 7 for Gamma_0(43) of weight 2 with␣
↪sign 0
```

The input must be valid or a ValueError is raised:

```
sage: M = CremonaModularSymbols(-1)
Traceback (most recent call last):
...
ValueError: the level (= -1) must be at least 2
sage: M = CremonaModularSymbols(0)
Traceback (most recent call last):
...
ValueError: the level (= 0) must be at least 2
```

The sign can only be 0 or 1 or -1:

```
sage: M = CremonaModularSymbols(10, sign = -2)
Traceback (most recent call last):
...
ValueError: sign (= -2) is not supported; use 0, +1 or -1
```

We do allow -1 as a sign (see trac ticket #9476):

```
sage: CremonaModularSymbols(10, sign = -1)
Cremona Modular Symbols space of dimension 0 for Gamma_0(10) of weight 2 with␣
↪sign -1
```

CHAPTER

SEVEN

RATIONAL RECONSTRUCTION

This file is a Cython implementation of rational reconstruction, using direct MPIR calls.

AUTHORS:

- ??? (2006 or before)
- Jeroen Demeyer (2014-10-20): move this function from `gmp.pxi`, simplify and fix some bugs, see trac ticket #17180

CHAPTER

EIGHT

RUBINSTEIN'S LCALC LIBRARY

This is a wrapper around Michael Rubinstein's lcalc. See http://oto.math.uwaterloo.ca/~mrubinst/L_function_public/CODE/.

AUTHORS:

- Rishikesh (2010): added compute_rank() and hardy_z_function()
- Yann Laigle-Chapuy (2009): refactored
- Rishikesh (2009): initial version

class `sage.libs.lcalc.lcalc_Lfunction.` **`Lfunction`**

Bases: `object`

Initialization of L-function objects. See derived class for details, this class is not supposed to be instantiated directly.

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: Lfunction_from_character(DirichletGroup(5)[1])
L-function with complex Dirichlet coefficients
```

`compute_rank` ( )

Computes the analytic rank (the order of vanishing at the center) of of the L-function

EXAMPLES:

```
sage: chi=DirichletGroup(5)[2] #This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L=Lfunction_from_character(chi, type="int")
sage: L.compute_rank()
0
sage: E=EllipticCurve([-82,0])
sage: L=Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: L.compute_rank()
3
```

`find_zeros` (*T1*, *T2*, *stepsize*)

Finds zeros on critical line between `T1` and `T2` using step size of stepsize. This function might miss zeros if step size is too large. This function computes the zeros of the L-function by using change in signs of areal valued function whose zeros coincide with the zeros of L-function.

Use *`find_zeros_via_N()`* for slower but more rigorous computation.

INPUT:

- `T1` – a real number giving the lower bound

- `T2` – a real number giving the upper bound
- `stepsize` – step size to be used for the zero search

OUTPUT:

list – A list of the imaginary parts of the zeros which were found.

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi=DirichletGroup(5)[2] #This is a quadratic character
sage: L=Lfunction_from_character(chi, type="int")
sage: L.find_zeros(5,15,.1)
[6.64845334472..., 9.83144443288..., 11.9588456260...]

sage: L=Lfunction_from_character(chi, type="double")
sage: L.find_zeros(1,15,.1)
[6.64845334472..., 9.83144443288..., 11.9588456260...]

sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi, type="complex")
sage: L.find_zeros(-8,8,.1)
[-4.13290370521..., 6.18357819545...]

sage: L=Lfunction_Zeta()
sage: L.find_zeros(10,29.1,.1)
[14.1347251417..., 21.0220396387..., 25.0108575801...]
```

find_zeros_via_N (*count=0*, *do_negative=False*, *max_refine=1025*, *rank=-1*, *test_explicit_formula=0*)

Finds `count` number of zeros with positive imaginary part starting at real axis. This function also verifies that all the zeros have been found.

INPUT:

- `count` - number of zeros to be found
- `do_negative` - (default: False) False to ignore zeros below the real axis.
- `max_refine` - when some zeros are found to be missing, the step size used to find zeros is refined. max_refine gives an upper limit on when lcalc should give up. Use default value unless you know what you are doing.
- `rank` - integer (default: -1) analytic rank of the L-function. If -1 is passed, then we attempt to compute it. (Use default if in doubt)
- `test_explicit_formula` - integer (default: 0) If nonzero, test the explicit fomula for additional confidence that all the zeros have been found and are accurate. This is still being tested, so using the default is recommended.

OUTPUT:

list – A list of the imaginary parts of the zeros that have been found

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi=DirichletGroup(5)[2] #This is a quadratic character
sage: L=Lfunction_from_character(chi, type="int")
sage: L.find_zeros_via_N(3)
[6.64845334472..., 9.83144443288..., 11.9588456260...]
```

```
sage: L=Lfunction_from_character(chi, type="double")
sage: L.find_zeros_via_N(3)
[6.64845334472..., 9.83144443288..., 11.9588456260...]

sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi, type="complex")
sage: L.find_zeros_via_N(3)
[6.18357819545..., 8.45722917442..., 12.6749464170...]

sage: L=Lfunction_Zeta()
sage: L.find_zeros_via_N(3)
[14.1347251417..., 21.0220396387..., 25.0108575801...]
```

hardy_z_function (*s*)

Computes the Hardy Z-function of the L-function at s

INPUT:

- `s` - a complex number with imaginary part between -0.5 and 0.5

EXAMPLES:

```
sage: chi=DirichletGroup(5)[2] #This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L=Lfunction_from_character(chi, type="int")
sage: L.hardy_z_function(0)
0.231750947504...
sage: L.hardy_z_function(.5).imag().abs() < 1.0e-16
True
sage: L.hardy_z_function(.4+.3*I)
0.2166144222685... - 0.00408187127850...*I
sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi,type="complex")
sage: L.hardy_z_function(0)
0.7939675904771...
sage: L.hardy_z_function(.5).imag().abs() < 1.0e-16
True
sage: E=EllipticCurve([-82,0])
sage: L=Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: L.hardy_z_function(2.1)
-0.00643179176869...
sage: L.hardy_z_function(2.1).imag().abs() < 1.0e-16
True
```

value (*s*, *derivative=0*)

Computes the value of the L-function at `s`

INPUT:

- `s` - a complex number
- `derivative` - integer (default: 0) the derivative to be evaluated
- `rotate` - (default: False) If True, this returns the value of the Hardy Z-function (sometimes called the Riemann-Siegel Z-function or the Siegel Z-function).

EXAMPLES:

```
sage: chi=DirichletGroup(5)[2] #This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L=Lfunction_from_character(chi, type="int")
```

```
sage: L.value(.5)  # abs tol 3e-15
0.231750947504016 + 5.75329642226136e-18*I
sage: L.value(.2+.4*I)
0.102558603193... + 0.190840777924...*I

sage: L=Lfunction_from_character(chi, type="double")
sage: L.value(.6)  # abs tol 3e-15
0.274633355856345 + 6.59869267328199e-18*I
sage: L.value(.6+I)
0.362258705721... + 0.433888250620...*I

sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi, type="complex")
sage: L.value(.5)
0.763747880117... + 0.216964767518...*I
sage: L.value(.6+5*I)
0.702723260619... - 1.10178575243...*I

sage: L=Lfunction_Zeta()
sage: L.value(.5)
-1.46035450880...
sage: L.value(.4+.5*I)
-0.450728958517... - 0.780511403019...*I
```

class `sage.libs.lcalc.lcalc_Lfunction.` **`Lfunction_C`**

Bases: *`sage.libs.lcalc.lcalc_Lfunction.Lfunction`*

The `Lfunction_C` class is used to represent L-functions with complex Dirichlet Coefficients. We assume that L-functions satisfy the following functional equation.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

$$\Lambda(s) = Q^s \left(\prod_{j=1}^{a} \Gamma(\kappa_j s + \gamma_j) \right) L(s)$$

See (23) in http://arxiv.org/abs/math/0412181

INPUT:

- `what_type_L` - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.
- `dirichlet_coefficient` - List of dirichlet coefficients of the L-function. Only first M coefficients are needed if they are periodic.
- `period` - If the coefficients are periodic, this should be the period of the coefficients.
- `Q` - See above
- `OMEGA` - See above
- `kappa` - List of the values of κ_j in the functional equation
- `gamma` - List of the values of γ_j in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

NOTES:

If an L-function satisfies $\Lambda(s) = \omega Q^s \Lambda(k-s)$, by replacing s by $s + (k-1)/2$, one can get it in the form we need.

class sage.libs.lcalc.lcalc_Lfunction. **Lfunction_D**

Bases: *sage.libs.lcalc.lcalc_Lfunction.Lfunction*

The Lfunction_D class is used to represent L-functions with real Dirichlet coefficients. We assume that L-functions satisfy the following functional equation.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

$$\Lambda(s) = Q^s \left(\prod_{j=1}^{a} \Gamma(\kappa_j s + \gamma_j) \right) L(s)$$

See (23) in http://arxiv.org/abs/math/0412181

INPUT:

- what_type_L - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.
- dirichlet_coefficient - List of dirichlet coefficients of the L-function. Only first M coefficients are needed if they are periodic.
- period - If the coefficients are periodic, this should be the period of the coefficients.
- Q - See above
- OMEGA - See above
- kappa - List of the values of κ_j in the functional equation
- gamma - List of the values of γ_j in the functional equation
- pole - List of the poles of L-function
- residue - List of the residues of the L-function

NOTES:

If an L-function satisfies $\Lambda(s) = \omega Q^s \Lambda(k-s)$, by replacing s by $s + (k-1)/2$, one can get it in the form we need.

class sage.libs.lcalc.lcalc_Lfunction. **Lfunction_I**

Bases: *sage.libs.lcalc.lcalc_Lfunction.Lfunction*

The Lfunction_I class is used to represent L-functions with integer Dirichlet Coefficients. We assume that L-functions satisfy the following functional equation.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

$$\Lambda(s) = Q^s \left(\prod_{j=1}^{a} \Gamma(\kappa_j s + \gamma_j) \right) L(s)$$

See (23) in http://arxiv.org/abs/math/0412181

INPUT:

- what_type_L - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.

- `dirichlet_coefficient` - List of dirichlet coefficients of the L-function. Only first M coefficients are needed if they are periodic.
- `period` - If the coefficients are periodic, this should be the period of the coefficients.
- `Q` - See above
- `OMEGA` - See above
- `kappa` - List of the values of κ_j in the functional equation
- `gamma` - List of the values of γ_j in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

NOTES:

If an L-function satisfies $\Lambda(s) = \omega Q^s \Lambda(k - s)$, by replacing s by $s + (k - 1)/2$, one can get it in the form we need.

class `sage.libs.lcalc.lcalc_Lfunction.`**`Lfunction_Zeta`**

Bases: *`sage.libs.lcalc.lcalc_Lfunction.Lfunction`*

The `Lfunction_Zeta` class is used to generate the Riemann zeta function.

`sage.libs.lcalc.lcalc_Lfunction.`**`Lfunction_from_character`**(*chi*, *type='complex'*)

Given a primitive Dirichlet character, this function returns an lcalc L-function object for the L-function of the character.

INPUT:

- `chi` - A Dirichlet character
- `use_type` - string (default: "complex") type used for the Dirichlet coefficients. This can be "int", "double" or "complex".

OUTPUT:

L-function object for `chi`.

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import Lfunction_from_character
sage: Lfunction_from_character(DirichletGroup(5)[1])
L-function with complex Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[2], type="int")
L-function with integer Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[2], type="double")
L-function with real Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[1], type="int")
Traceback (most recent call last):
...
ValueError: For non quadratic characters you must use type="complex"
```

`sage.libs.lcalc.lcalc_Lfunction.`**`Lfunction_from_elliptic_curve`**(*E*, *number_of_coeffs=10000*)

Given an elliptic curve E, return an L-function object for the function $L(s, E)$.

INPUT:

- `E` - An elliptic curve

- `number_of_coeffs` - integer (default: 10000) The number of coefficients to be used when constructing the L-function object. Right now this is fixed at object creation time, and is not automatically set intelligently.

OUTPUT:

L-function object for `L(s,E)` .

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import Lfunction_from_elliptic_curve
sage: L = Lfunction_from_elliptic_curve(EllipticCurve('37'))
sage: L
L-function with real Dirichlet coefficients
sage: L.value(0.5).abs() < 1e-15   # "noisy" zero on some platforms (see #9615)
True
sage: L.value(0.5, derivative=1)
0.305999...
```

CHAPTER

NINE

HYPERELLIPTIC CURVE POINT FINDING, VIA RATPOINTS.

`sage.libs.ratpoints.ratpoints` (*coeffs*, *H*, *verbose=False*, *max=0*, *min_x_denom=None*, *max_x_denom=None*, *intervals=[]*)

Access the ratpoints library to find points on the hyperelliptic curve:

$y^2 = a_n x^n + \cdots + a_1 x + a_0.$

INPUT:

- `coeffs` – list of integer coefficients a_0 , a_1, ..., a_n
- `H` – the bound for the denominator and the absolute value of the numerator of the x-coordinate
- `verbose` – if `True` , ratpoints will print comments about its progress
- `max` – maximum number of points to find (if 0, find all of them)

OUTPUT:

The points output by this program are points in (1, ceil(n/2), 1)-weighted projective space. If n is even, then the associated homogeneous equation is $y^2 = a_n x^n + \cdots + a_1 x z^{n-1} + a_0 z^n$ while if n is odd, it is $y^2 = a_n x^n z + \cdots + a_1 x z^n + a_0 z^{n+1}$.

EXAMPLE:

```
sage: from sage.libs.ratpoints import ratpoints
sage: for x,y,z in ratpoints([1..6], 200):
....:     print(-1*y^2 + 1*z^6 + 2*x*z^5 + 3*x^2*z^4 + 4*x^3*z^3 + 5*x^4*z^2 +␣
↪6*x^5*z)
0
0
0
0
0
0
0
sage: for x,y,z in ratpoints([1..5], 200):
....:    print(-1*y^2 + 1*z^4 + 2*x*z^3 + 3*x^2*z^2 + 4*x^3*z + 5*x^4)
0
0
0
0
0
0
0
0

sage: for x,y,z in ratpoints([1..200], 1000):
....:    print("{} {} {}".format(x,y,z))
```

```
1 0 0
0 1 1
0 -1 1
201␣
↪2535301200456458802993406410752000000000000000000000000000000000000000000000000000000000000000000
↪200
201 -
↪2535301200456458802993406410752000000000000000000000000000000000000000000000000000000000000000000
↪200
```

The denominator of x can be restricted, for example to find integral points:

```
sage: from sage.libs.ratpoints import ratpoints
sage: coeffs = [400, -112, 0, 1]
sage: ratpoints(coeffs, 10^6, max_x_denom=1, intervals=[[-10,0],[1000,2000]])
[(1, 0, 0), (-8, 28, 1), (-8, -28, 1), (-7, 29, 1), (-7, -29, 1),
 (-4, 28, 1), (-4, -28, 1), (0, 20, 1), (0, -20, 1), (1368, 50596, 1),
 (1368, -50596, 1), (1624, 65444, 1), (1624, -65444, 1)]

sage: ratpoints(coeffs, 1000, min_x_denom=100, max_x_denom=200)
[(1, 0, 0),
(-656, 426316, 121),
(-656, -426316, 121),
(452, 85052, 121),
(452, -85052, 121),
(988, 80036, 121),
(988, -80036, 121),
(-556, 773188, 169),
(-556, -773188, 169),
(264, 432068, 169),
(264, -432068, 169)]
```

Finding the integral points on the compact component of an elliptic curve:

```
sage: E = EllipticCurve([0,1,0,-35220,-1346400])
sage: e1, e2, e3 = E.division_polynomial(2).roots(multiplicities=False)
sage: coeffs = [E.a6(),E.a4(),E.a2(),1]
sage: ratpoints(coeffs, 1000, max_x_denom=1, intervals=[[e3,e2]])
[(1, 0, 0),
(-165, 0, 1),
(-162, 366, 1),
(-162, -366, 1),
(-120, 1080, 1),
(-120, -1080, 1),
(-90, 1050, 1),
(-90, -1050, 1),
(-85, 1020, 1),
(-85, -1020, 1),
(-42, 246, 1),
(-42, -246, 1),
(-40, 0, 1)]
```

CHAPTER

TEN

LIBSINGULAR: FUNCTIONS

Sage implements a C wrapper around the Singular interpreter which allows to call any function directly from Sage without string parsing or interprocess communication overhead. Users who do not want to call Singular functions directly, usually do not have to worry about this interface, since it is handled by higher level functions in Sage.

AUTHORS:

- Michael Brickenstein (2009-07): initial implementation, overall design
- Martin Albrecht (2009-07): clean up, enhancements, etc.
- Michael Brickenstein (2009-10): extension to more Singular types
- Martin Albrecht (2010-01): clean up, support for attributes
- Simon King (2011-04): include the documentation provided by Singular as a code block.
- Burcin Erocal, Michael Brickenstein, Oleksandr Motsak, Alexander Dreyer, Simon King (2011-09) plural support

EXAMPLES:

The direct approach for loading a Singular function is to call the function `singular_function()` with the function name as parameter:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<a,b,c,d> = PolynomialRing(GF(7))
sage: std = singular_function('std')
sage: I = sage.rings.ideal.Cyclic(P)
sage: std(I)
[a + b + c + d,
 b^2 + 2*b*d + d^2,
 b*c^2 + c^2*d - b*d^2 - d^3,
 b*c*d^2 + c^2*d^2 - b*d^3 + c*d^3 - d^4 - 1,
 b*d^4 + d^5 - b - d,
 c^3*d^2 + c^2*d^3 - c - d,
 c^2*d^4 + b*c - b*d + c*d - 2*d^2]
```

If a Singular library needs to be loaded before a certain function is available, use the `lib()` function as shown below:

```
sage: from sage.libs.singular.function import singular_function, lib as singular_lib
sage: primdecSY = singular_function('primdecSY')
Traceback (most recent call last):
...
NameError: Function 'primdecSY' is not defined.

sage: singular_lib('primdec.lib')
sage: primdecSY = singular_function('primdecSY')
```

There is also a short-hand notation for the above:

```
sage: import sage.libs.singular.function_factory
sage: primdecSY = sage.libs.singular.function_factory.ff.primdec__lib.primdecSY
```

The above line will load "primdec.lib" first and then load the function `primdecSY`.

TESTS:

```
sage: from sage.libs.singular.function import singular_function
sage: std = singular_function('std')
sage: loads(dumps(std)) == std
True
```

class `sage.libs.singular.function.`**`BaseCallHandler`**

Bases: `object`

A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.

class `sage.libs.singular.function.`**`Converter`**

Bases: `sage.structure.sage_object.SageObject`

A *`Converter`* interfaces between Sage objects and Singular interpreter objects.

`ring` ( )

Return the ring in which the arguments of this list live.

EXAMPLE:

```
sage: from sage.libs.singular.function import Converter
sage: P.<a,b,c> = PolynomialRing(GF(127))
sage: Converter([a,b,c],ring=P).ring()
Multivariate Polynomial Ring in a, b, c over Finite Field of size 127
```

class `sage.libs.singular.function.`**`KernelCallHandler`**

Bases: *`sage.libs.singular.function.BaseCallHandler`*

A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.

This class implements calling a kernel function.

Note: Do not construct this class directly, use *`singular_function()`* instead.

class `sage.libs.singular.function.`**`LibraryCallHandler`**

Bases: *`sage.libs.singular.function.BaseCallHandler`*

A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.

This class implements calling a library function.

Note: Do not construct this class directly, use *`singular_function()`* instead.

class `sage.libs.singular.function.`**`Resolution`**

Bases: `object`

A simple wrapper around Singular's resolutions.

class sage.libs.singular.function.**RingWrap**

Bases: object

A simple wrapper around Singular's rings.

characteristic()

Get characteristic.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).characteristic()
0
```

is_commutative()

Determine whether a given ring is commutative.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).is_commutative()
True
```

ngens()

Get number of generators.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).ngens()
3
```

npars()

Get number of parameters.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).npars()
0
```

ordering_string()

Get Singular string defining monomial ordering.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).ordering_string()
'dp(3),C'
```

par_names ( )

Get parameter names.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).par_names()
[]
```

var_names ( )

Get names of variables.

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).var_names()
['x', 'y', 'z']
```

class sage.libs.singular.function.**SingularFunction**

Bases: sage.structure.sage_object.SageObject

The base class for Singular functions either from the kernel or from the library.

class sage.libs.singular.function.**SingularKernelFunction**

Bases: *sage.libs.singular.function.SingularFunction*

EXAMPLES:

```
sage: from sage.libs.singular.function import SingularKernelFunction
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(x, x+1)
sage: f = SingularKernelFunction("std")
sage: f(I)
[1]
```

class sage.libs.singular.function.**SingularLibraryFunction**

Bases: *sage.libs.singular.function.SingularFunction*

EXAMPLES:

```
sage: from sage.libs.singular.function import SingularLibraryFunction
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(x, x+1)
sage: f = SingularLibraryFunction("groebner")
```

```
sage: f(I)
[1]
```

sage.libs.singular.function.**all_singular_poly_wrapper**(*s*)

Tests for a sequence `s`, whether it consists of singular polynomials.

EXAMPLE:

```
sage: from sage.libs.singular.function import all_singular_poly_wrapper
sage: P.<x,y,z> = QQ[]
sage: all_singular_poly_wrapper([x+1, y])
True
sage: all_singular_poly_wrapper([x+1, y, 1])
False
```

sage.libs.singular.function.**all_vectors**(*s*)

Checks if a sequence `s` consists of free module elements over a singular ring.

EXAMPLE:

```
sage: from sage.libs.singular.function import all_vectors
sage: P.<x,y,z> = QQ[]
sage: M = P**2
sage: all_vectors([x])
False
sage: all_vectors([(x,y)])
False
sage: all_vectors([M(0), M((x,y))])
True
sage: all_vectors([M(0), M((x,y)),(0,0)])
False
```

sage.libs.singular.function.**is_sage_wrapper_for_singular_ring**(*ring*)

Check whether wrapped ring arises from Singular or Singular/Plural.

EXAMPLE:

```
sage: from sage.libs.singular.function import is_sage_wrapper_for_singular_ring
sage: P.<x,y,z> = QQ[]
sage: is_sage_wrapper_for_singular_ring(P)
True
```

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: is_sage_wrapper_for_singular_ring(P)
True
```

sage.libs.singular.function.**is_singular_poly_wrapper**(*p*)

Checks if p is some data type corresponding to some singular `poly`.

EXAMPLE:

```
sage: from sage.libs.singular.function import is_singular_poly_wrapper
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: is_singular_poly_wrapper(x+y)
True
```

sage.libs.singular.function.**lib**(*name*)

Load the Singular library `name`.

INPUT:

- `name` – a Singular library name

EXAMPLE:

```
sage: from sage.libs.singular.function import singular_function
sage: from sage.libs.singular.function import lib as singular_lib
sage: singular_lib('general.lib')
sage: primes = singular_function('primes')
sage: primes(2,10, ring=GF(127)['x,y,z'])
(2, 3, 5, 7)
```

sage.libs.singular.function.**list_of_functions**(*packages=False*)

Return a list of all function names currently available.

INPUT:

- `packages` - include local functions in packages.

EXAMPLE:

```
sage: 'groebner' in sage.libs.singular.function.list_of_functions()
True
```

sage.libs.singular.function.**singular_function**(*name*)

Construct a new libSingular function object for the given `name`.

This function works both for interpreter and built-in functions.

INPUT:

- `name` – the name of the function

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: f = 3*x*y + 2*z + 1
sage: g = 2*x + 1/2
sage: I = Ideal([f,g])
```

```
sage: from sage.libs.singular.function import singular_function
sage: std = singular_function("std")
sage: std(I)
[3*y - 8*z - 4, 4*x + 1]
sage: size = singular_function("size")
sage: size([2, 3, 3])
3
sage: size("sage")
4
sage: size(["hello", "sage"])
2
sage: factorize = singular_function("factorize")
sage: factorize(f)
[[1, 3*x*y + 2*z + 1], (1, 1)]
sage: factorize(f, 1)
[3*x*y + 2*z + 1]
```

We give a wrong number of arguments:

```
sage: factorize()
Traceback (most recent call last):
...
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
sage: factorize(f, 1, 2)
Traceback (most recent call last):
...
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
sage: factorize(f, 1, 2, 3)
Traceback (most recent call last):
...
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
```

The Singular function `list` can be called with any number of arguments:

```
sage: singular_list = singular_function("list")
sage: singular_list(2, 3, 6)
[2, 3, 6]
sage: singular_list()
[]
sage: singular_list(1)
[1]
sage: singular_list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We try to define a non-existing function:

```
sage: number_foobar = singular_function('number_foobar');
Traceback (most recent call last):
...
NameError: Function 'number_foobar' is not defined.
```

```
sage: from sage.libs.singular.function import lib as singular_lib
sage: singular_lib('general.lib')
sage: number_e = singular_function('number_e')
sage: number_e(10r)
67957045707/25000000000
sage: RR(number_e(10r))
2.71828182828000
```

```
sage: singular_lib('primdec.lib')
sage: primdecGTZ = singular_function("primdecGTZ")
sage: primdecGTZ(I)
[[[y - 8/3*z - 4/3, x + 1/4], [y - 8/3*z - 4/3, x + 1/4]]]
sage: singular_list((1,2,3),3,[1,2,3], ring=P)
[(1, 2, 3), 3, [1, 2, 3]]
sage: ringlist=singular_function("ringlist")
sage: l = ringlist(P)
sage: l[3].__class__
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic
↪'>
sage: l
[0, ['x', 'y', 'z'], [['dp', (1, 1, 1)], ['C', (0,)]], [0]]
```

```
sage: ring=singular_function("ring")
sage: ring(l)
<RingWrap>
sage: matrix = Matrix(P,2,2)
sage: matrix.randomize(terms=1)
sage: det = singular_function("det")
sage: det(matrix)
-3/5*x*y*z
sage: coeffs = singular_function("coeffs")
sage: coeffs(x*y+y+1,y)
[    1]
[x + 1]
sage: intmat = Matrix(ZZ, 2,2, [100,2,3,4])
sage: det(intmat)
394
sage: random = singular_function("random")
sage: A = random(10,2,3); A.nrows(), max(A.list()) <= 10
(2, True)
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: M=P**3
sage: leadcoef = singular_function("leadcoef")
sage: v=M((100*x,5*y,10*z*x*y))
sage: leadcoef(v)
10
sage: v = M([x+y,x*y+y**3,z])
sage: lead = singular_function("lead")
sage: lead(v)
(0, y^3)
sage: jet = singular_function("jet")
sage: jet(v, 2)
(x + y, x*y, z)
sage: syz = singular_function("syz")
sage: I = P.ideal([x+y,x*y-y, y*2,x**2+1])
sage: M = syz(I)
sage: M
[(-2*y, 2, y + 1, 0), (0, -2, x - 1, 0), (x*y - y, -y + 1, 1, -y), (x^2 + 1, -x -
↪1, -1, -x)]
sage: singular_lib("mprimdec.lib")
sage: syz(M)
[(-x - 1, y - 1, 2*x, -2*y)]
sage: GTZmod = singular_function("GTZmod")
sage: GTZmod(M)
[[[(-2*y, 2, y + 1, 0), (0, x + 1, 1, -y), (0, -2, x - 1, 0), (x*y - y, -y + 1,
↪1, -y), (x^2 + 1, 0, 0, -x - y)], [0]]]
sage: mres = singular_function("mres")
sage: resolution = mres(M, 0)
sage: resolution
<Resolution>
sage: singular_list(resolution)
[[(-2*y, 2, y + 1, 0), (0, -2, x - 1, 0), (x*y - y, -y + 1, 1, -y), (x^2 + 1, -x -
↪ 1, -1, -x)], [(-x - 1, y - 1, 2*x, -2*y)], [(0)]]

sage: A.<x,y> = FreeAlgebra(QQ, 2)
sage: P.<x,y> = A.g_algebra({y*x:-x*y})
sage: I= Sequence([x*y,x+y], check=False, immutable=True)
sage: twostd = singular_function("twostd")
sage: twostd(I)
[x + y, y^2]
```

```
sage: M=syz(I)
doctest...
sage: M
[(x + y, x*y)]
sage: syz(M)
[(0)]
sage: mres(I, 0)
<Resolution>
sage: M=P**3
sage: v=M((100*x,5*y,10*y*x*y))
sage: leadcoef(v)
-10
sage: v = M([x+y,x*y+y**3,x])
sage: lead(v)
(0, y^3)
sage: jet(v, 2)
(x + y, x*y, x)
sage: l = ringlist(P)
sage: len(l)
6
sage: ring(l)
<noncommutative RingWrap>
sage: I=twostd(I)
sage: l[3]=I
sage: ring(l)
<noncommutative RingWrap>
```

CHAPTER

ELEVEN

LIBSINGULAR: FUNCTION FACTORY

AUTHORS:

- Martin Albrecht (2010-01): initial version

class sage.libs.singular.function_factory.**SingularFunctionFactory**

Bases: object

A convenient interface to libsingular functions.

trait_names()

EXAMPLE:

```
sage: import sage.libs.singular.function_factory
sage: "groebner" in sage.libs.singular.function_factory.ff.trait_names()
True
```

CHAPTER

TWELVE

LIBSINGULAR: CONVERSION ROUTINES AND INITIALISATION

AUTHOR:

- Martin Albrecht <malb@informatik.uni-bremen.de>

CHAPTER THIRTEEN

WRAPPER FOR SINGULAR'S POLYNOMIAL ARITHMETIC

AUTHOR:

- Martin Albrecht (2009-07): refactoring

CHAPTER

FOURTEEN

LIBSINGULAR: OPTIONS

Singular uses a set of global options to determine verbosity and the behavior of certain algorithms. We provide an interface to these options in the most 'natural' python-ic way. Users who do not wish to deal with Singular functions directly usually do not have to worry about this interface or Singular options in general since this is taken care of by higher level functions.

We compute a Groebner basis for Cyclic-5 in two different contexts:

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(127))
sage: I = sage.rings.ideal.Cyclic(P)
sage: import sage.libs.singular.function_factory
sage: std = sage.libs.singular.function_factory.ff.std
```

By default, tail reductions are performed:

```
sage: from sage.libs.singular.option import opt, opt_ctx
sage: opt['red_tail']
True
sage: std(I)[-1]
d^2*e^6 + 28*b*c*d + ...
```

If we don't want this, we can create an option context, which disables this:

```
sage: with opt_ctx(red_tail=False, red_sb=False):
...       std(I)[-1]
d^2*e^6 + 8*c^3 + ...
```

However, this does not affect the global state:

```
sage: opt['red_tail']
True
```

On the other hand, any assignment to an option object will immediately change the global state:

```
sage: opt['red_tail'] = False
sage: opt['red_tail']
False
sage: opt['red_tail'] = True
sage: opt['red_tail']
True
```

Assigning values within an option context, only affects this context:

```
sage: with opt_ctx:
...       opt['red_tail'] = False
```

```
sage: opt['red_tail']
True
```

Option contexts can also be safely stacked:

```
sage: with opt_ctx:
....:     opt['red_tail'] = False
....:     print(opt)
....:     with opt_ctx:
....:         opt['red_through'] = False
....:         print(opt)
...
general options for libSingular (current value 0x00000082)
general options for libSingular (current value 0x00000002)

sage: print(opt)
general options for libSingular (current value 0x02000082)
```

Furthermore, the integer valued options `deg_bound` and `mult_bound` can be used:

```
sage: R.<x,y> = QQ[]
sage: I = R*[x^3+y^2,x^2*y+1]
sage: opt['deg_bound'] = 2
sage: std(I)
[x^2*y + 1, x^3 + y^2]
sage: opt['deg_bound'] = 0
sage: std(I)
[y^3 - x, x^2*y + 1, x^3 + y^2]
```

The same interface is available for verbosity options:

```
sage: from sage.libs.singular.option import opt_verb
sage: opt_verb['not_warn_sb']
False
sage: opt.reset_default()  # needed to avoid side effects
sage: opt_verb.reset_default()  # needed to avoid side effects
```

AUTHOR:

- Martin Albrecht (2009-08): initial implementation
- Martin Albrecht (2010-01): better interface, verbosity options
- Simon King (2010-07): Python-ic option names; deg_bound and mult_bound

class `sage.libs.singular.option.`**`LibSingularOptions`**

Bases: *`sage.libs.singular.option.LibSingularOptions_abstract`*

Pythonic Interface to libSingular's options.

Supported options are:

- `return_sb` or `returnSB` - the functions `syz`, `intersect`, `quotient`, `modulo` return a standard base instead of a generating set if `return_sb` is set. This option should not be used for `lift`.
- `fast_hc` or `fastHC` - tries to find the highest corner of the staircase (HC) as fast as possible during a standard basis computation (only used for local orderings).

- `int_strategy` or `intStrategy` - avoids division of coefficients during standard basis computations. This option is ring dependent. By default, it is set for rings with characteristic 0 and not set for all other rings.
- `lazy` - uses a more lazy approach in std computations, which was used in SINGULAR version before 2-0 (and which may lead to faster or slower computations, depending on the example).
- `length` - select shorter reducers in std computations.
- `not_regularity` or `notRegularity` - disables the regularity bound for `res` and `mres` .
- `not_sugar` or `notSugar` - disables the sugar strategy during standard basis computation.
- `not_buckets` or `notBuckets` - disables the bucket representation of polynomials during standard basis computations. This option usually decreases the memory usage but increases the computation time. It should only be set for memory-critical standard basis computations.
- `old_std` or `oldStd` - uses a more lazy approach in std computations, which was used in SINGULAR version before 2-0 (and which may lead to faster or slower computations, depending on the example).
- `prot` - shows protocol information indicating the progress during the following computations: `facstd` , `fglm` , `groebner` , `lres` , `mres` , `minres` , `mstd` , `res` , `slimgb` , `sres` , `std` , `stdfglm` , `stdhilb` , `syz` .
- $red_s b$‘ or `redSB` - computes a reduced standard basis in any standard basis computation.
- `red_tail` or `redTail` - reduction of the tails of polynomials during standard basis computations. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.
- `red_through` or `redThrough` - for inhomogenous input, polynomial reductions during standard basis computations are never postponed, but always finished through. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.
- `sugar_crit` or `sugarCrit` - uses criteria similar to the homogeneous case to keep more useless pairs.
- `weight_m` or `weightM` - automatically computes suitable weights for the weighted ecart and the weighted sugar method.

In addition, two integer valued parameters are supported, namely:

- `deg_bound` or `degBound` - The standard basis computation is stopped if the total (weighted) degree exceeds `deg_bound` . `deg_bound` should not be used for a global ordering with inhomogeneous input. Reset this bound by setting `deg_bound` to 0. The exact meaning of “degree” depends on the ring odering and the command: `slimgb` uses always the total degree with weights 1, `std` does so for block orderings, only.
- `mult_bound` or `multBound` - The standard basis computation is stopped if the ideal is zero-dimensional in a ring with local ordering and its multiplicity is lower than `mult_bound` . Reset this bound by setting `mult_bound` to 0.

EXAMPLE:

```
sage: from sage.libs.singular.option import LibSingularOptions
sage: libsingular_options = LibSingularOptions()
sage: libsingular_options
general options for libSingular (current value 0x06000082)
```

Here we demonstrate the intended way of using libSingular options:

```
sage: R.<x,y> = QQ[]
sage: I = R*[x^3+y^2,x^2*y+1]
sage: I.groebner_basis(deg_bound=2)
[x^3 + y^2, x^2*y + 1]
sage: I.groebner_basis()
[x^3 + y^2, x^2*y + 1, y^3 - x]
```

The option `mult_bound` is only relevant in the local case:

```
sage: from sage.libs.singular.option import opt
sage: Rlocal.<x,y,z> = PolynomialRing(QQ, order='ds')
sage: x^2<x
True
sage: J = [x^7+y^7+z^6,x^6+y^8+z^7,x^7+y^5+z^8, x^2*y^3+y^2*z^3+x^3*z^2,x^3*y^2+y^
↪3*z^2+x^2*z^3]*Rlocal
sage: J.groebner_basis(mult_bound=100)
[x^3*y^2 + y^3*z^2 + x^2*z^3, x^2*y^3 + x^3*z^2 + y^2*z^3, y^5, x^6 + x*y^4*z^5,␣
↪x^4*z^2 - y^4*z^2 - x^2*y*z^3 + x*y^2*z^3, z^6 - x*y^4*z^4 - x^3*y*z^5]
sage: opt['red_tail'] = True # the previous commands reset opt['red_tail'] to␣
↪False
sage: J.groebner_basis()
[x^3*y^2 + y^3*z^2 + x^2*z^3, x^2*y^3 + x^3*z^2 + y^2*z^3, y^5, x^6, x^4*z^2 - y^
↪4*z^2 - x^2*y*z^3 + x*y^2*z^3, z^6, y^4*z^3 - y^3*z^4 - x^2*z^5, x^3*y*z^4 - x^
↪2*y^2*z^4 + x*y^3*z^4, x^3*z^5, x^2*y*z^5 + y^3*z^5, x*y^3*z^5]
```

reset_default ( )

Reset libSingular's default options.

EXAMPLE:

```
sage: from sage.libs.singular.option import opt
sage: opt['red_tail']
True
sage: opt['red_tail'] = False
sage: opt['red_tail']
False
sage: opt['deg_bound']
0
sage: opt['deg_bound'] = 2
sage: opt['deg_bound']
2
sage: opt.reset_default()
sage: opt['red_tail']
True
sage: opt['deg_bound']
0
```

class `sage.libs.singular.option.`**LibSingularOptionsContext**

Bases: `object`

Option context

This object localizes changes to options.

EXAMPLE:

```
sage: from sage.libs.singular.option import opt, opt_ctx
sage: opt
general options for libSingular (current value 0x06000082)
```

```
sage: with opt_ctx(redTail=False):
....:     print(opt)
....:     with opt_ctx(redThrough=False):
....:         print(opt)
general options for libSingular (current value 0x04000082)
general options for libSingular (current value 0x04000002)

sage: print(opt)
general options for libSingular (current value 0x06000082)
```

opt

class sage.libs.singular.option.**LibSingularOptions_abstract**

Bases: object

Abstract Base Class for libSingular options.

load (*value=None*)

EXAMPLE:

```
sage: from sage.libs.singular.option import opt as sopt
sage: bck = sopt.save(); hex(bck[0]), bck[1], bck[2]
('0x6000082', 0, 0)
sage: sopt['redTail'] = False
sage: hex(int(sopt))
'0x4000082'
sage: sopt.load(bck)
sage: sopt['redTail']
True
```

save ()

Return a triple of integers that allow reconstruction of the options.

EXAMPLE:

```
sage: from sage.libs.singular.option import opt
sage: opt['deg_bound']
0
sage: opt['red_tail']
True
sage: s = opt.save()
sage: opt['deg_bound'] = 2
sage: opt['red_tail'] = False
sage: opt['deg_bound']
2
sage: opt['red_tail']
False
sage: opt.load(s)
sage: opt['deg_bound']
0
sage: opt['red_tail']
True
sage: opt.reset_default()  # needed to avoid side effects
```

class sage.libs.singular.option.**LibSingularVerboseOptions**

Bases: *sage.libs.singular.option.LibSingularOptions_abstract*

Pythonic Interface to libSingular's verbosity options.

Supported options are:

- `mem` - shows memory usage in square brackets.
- `yacc` - Only available in debug version.
- `redefine` - warns about variable redefinitions.
- `reading` - shows the number of characters read from a file.
- `loadLib` or `load_lib` - shows loading of libraries.
- `debugLib` or `debug_lib` - warns about syntax errors when loading a library.
- `loadProc` or `load_proc` - shows loading of procedures from libraries.
- `defRes` or `def_res` - shows the names of the syzygy modules while converting `resolution` to `list`.
- `usage` - shows correct usage in error messages.
- `Imap` or `imap` - shows the mapping of variables with the `fetch` and `imap` commands.
- `notWarnSB` or `not_warn_sb` - do not warn if a basis is not a standard basis
- `contentSB` or `content_sb` - avoids to divide by the content of a polynomial in `std` and related algorithms. Should usually not be used.
- `cancelunit` - avoids to divide polynomials by non-constant units in `std` in the local case. Should usually not be used.

EXAMPLE:

```
sage: from sage.libs.singular.option import LibSingularVerboseOptions
sage: libsingular_verbose = LibSingularVerboseOptions()
sage: libsingular_verbose
verbosity options for libSingular (current value 0x00002851)
```

reset_default ()

Return to libSingular's default verbosity options

EXAMPLE:

```
sage: from sage.libs.singular.option import opt_verb
sage: opt_verb['not_warn_sb']
False
sage: opt_verb['not_warn_sb'] = True
sage: opt_verb['not_warn_sb']
True
sage: opt_verb.reset_default()
sage: opt_verb['not_warn_sb']
False
```

CHAPTER

FIFTEEN

WRAPPER FOR SINGULAR'S RINGS

AUTHORS:

- Martin Albrecht (2009-07): initial implementation
- Kwankyu Lee (2010-06): added matrix term order support

sage.libs.singular.ring. **currRing_wrapper** ( )

Returns a wrapper for the current ring, for use in debugging ring_refcount_dict.

EXAMPLES:

```
sage: from sage.libs.singular.ring import currRing_wrapper
sage: currRing_wrapper()
The ring pointer ...
```

sage.libs.singular.ring. **poison_currRing** (*frame*, *event*, *arg*)

Poison the `currRing` pointer.

This function sets the `currRing` to an illegal value. By setting it as the python debug hook, you can poison the currRing before every evaluated Python command (but not within Cython code).

INPUT:

- `frame` , `event` , `arg` – the standard arguments for the CPython debugger hook. They are not used.

OUTPUT:

Returns itself, which ensures that `poison_currRing()` will stay in the debugger hook.

EXAMPLES:

```
sage: previous_trace_func = sys.gettrace()   # None if no debugger running
sage: from sage.libs.singular.ring import poison_currRing
sage: sys.settrace(poison_currRing)
sage: sys.gettrace()
<built-in function poison_currRing>
sage: sys.settrace(previous_trace_func)  # switch it off again
```

sage.libs.singular.ring. **print_currRing** ( )

Print the `currRing` pointer.

EXAMPLES:

```
sage: from sage.libs.singular.ring import print_currRing
sage: print_currRing()   # random output
DEBUG: currRing == 0x7fc6fa6ec480

sage: from sage.libs.singular.ring import poison_currRing
```

```
sage: _ = poison_currRing(None, None, None)
sage: print_currRing()
DEBUG: currRing == 0x0
```

class `sage.libs.singular.ring.` **`ring_wrapper_Py`**

Bases: `object`

Python object wrapping the ring pointer.

This is useful to store ring pointers in Python containers.

You must not construct instances of this class yourself, use `wrap_ring()` instead.

EXAMPLES:

```
sage: from sage.libs.singular.ring import ring_wrapper_Py
sage: ring_wrapper_Py
<type 'sage.libs.singular.ring.ring_wrapper_Py'>
```

CHAPTER

SIXTEEN

SINGULAR'S GROEBNER STRATEGY OBJECTS

AUTHORS:

- Martin Albrecht (2009-07): initial implementation
- Michael Brickenstein (2009-07): initial implementation
- Hans Schoenemann (2009-07): initial implementation

class `sage.libs.singular.groebner_strategy.`**`GroebnerStrategy`**

Bases: `sage.structure.sage_object.SageObject`

A Wrapper for Singular's Groebner Strategy Object.

This object provides functions for normal form computations and other functions for Groebner basis computation.

ALGORITHM:

Uses Singular via libSINGULAR

`ideal` ( )

Return the ideal this strategy object is defined for.

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(GF(32003))
sage: I = Ideal([x + z, y + z])
sage: strat = GroebnerStrategy(I)
sage: strat.ideal()
Ideal (x + z, y + z) of Multivariate Polynomial Ring in x, y, z over Finite␣
↪Field of size 32003
```

`normal_form` (*p*)

Compute the normal form of `p` with respect to the generators of this object.

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x + z, y + z])
sage: strat = GroebnerStrategy(I)
sage: strat.normal_form(x*y) # indirect doctest
z^2
sage: strat.normal_form(x + 1)
-z + 1
```

TESTS:

```
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([P(0)])
sage: strat = GroebnerStrategy(I)
sage: strat.normal_form(x)
x
sage: strat.normal_form(P(0))
0
```

ring ( )

Return the ring this strategy object is defined over.

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(GF(32003))
sage: I = Ideal([x + z, y + z])
sage: strat = GroebnerStrategy(I)
sage: strat.ring()
Multivariate Polynomial Ring in x, y, z over Finite Field of size 32003
```

class sage.libs.singular.groebner_strategy.**NCGroebnerStrategy**

Bases: sage.structure.sage_object.SageObject

A Wrapper for Singular's Groebner Strategy Object.

This object provides functions for normal form computations and other functions for Groebner basis computation.

ALGORITHM:

Uses Singular via libSINGULAR

ideal ( )

Return the ideal this strategy object is defined for.

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: strat.ideal() == I
True
```

normal_form (*p*)

Compute the normal form of p with respect to the generators of this object.

EXAMPLE:

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: JL = H.ideal([x^3, y^3, z^3 - 4*z])
sage: JT = H.ideal([x^3, y^3, z^3 - 4*z], side='twosided')
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: SL = NCGroebnerStrategy(JL.std())
sage: ST = NCGroebnerStrategy(JT.std())
sage: SL.normal_form(x*y^2)
x*y^2
```

```
sage: ST.normal_form(x*y^2)
y*z
```

ring ( )

Return the ring this strategy object is defined over.

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: strat.ring() is H
True
```

sage.libs.singular.groebner_strategy.**unpickle_GroebnerStrategy0** (*I*)

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(GF(32003))
sage: I = Ideal([x + z, y + z])
sage: strat = GroebnerStrategy(I)
sage: loads(dumps(strat)) == strat # indirect doctest
True
```

sage.libs.singular.groebner_strategy.**unpickle_NCGroebnerStrategy0** (*I*)

EXAMPLE:

```
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: loads(dumps(strat)) == strat   # indirect doctest
True
```

CHAPTER

SEVENTEEN

CYTHON WRAPPER FOR THE PARMA POLYHEDRA LIBRARY (PPL)

The Parma Polyhedra Library (PPL) is a library for polyhedral computations over **Q**. This interface tries to reproduce the C++ API as faithfully as possible in Cython/Sage. For example, the following C++ excerpt:

```
Variable x(0);
Variable y(1);
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 3);
cs.insert(y >= 0);
cs.insert(y <= 3);
C_Polyhedron poly_from_constraints(cs);
```

translates into:

```
sage: from sage.libs.ppl import Variable, Constraint_System, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert(x >= 0)
sage: cs.insert(x <= 3)
sage: cs.insert(y >= 0)
sage: cs.insert(y <= 3)
sage: poly_from_constraints = C_Polyhedron(cs)
```

The same polyhedron constructed from generators:

```
sage: from sage.libs.ppl import Variable, Generator_System, C_Polyhedron, point
sage: gs = Generator_System()
sage: gs.insert(point(0*x + 0*y))
sage: gs.insert(point(0*x + 3*y))
sage: gs.insert(point(3*x + 0*y))
sage: gs.insert(point(3*x + 3*y))
sage: poly_from_generators = C_Polyhedron(gs)
```

Rich comparisons test equality/inequality and strict/non-strict containment:

```
sage: poly_from_generators == poly_from_constraints
True
sage: poly_from_generators >= poly_from_constraints
True
sage: poly_from_generators <  poly_from_constraints
False
sage: poly_from_constraints.minimized_generators()
Generator_System {point(0/1, 0/1), point(0/1, 3/1), point(3/1, 0/1), point(3/1, 3/1)}
```

```
sage: poly_from_constraints.minimized_constraints()
Constraint_System {-x0+3>=0, -x1+3>=0, x0>=0, x1>=0}
```

As we see above, the library is generally easy to use. There are a few pitfalls that are not entirely obvious without consulting the documentation, in particular:

- There are no vectors used to describe `Generator` (points, closure points, rays, lines) or `Constraint` (strict inequalities, non-strict inequalities, or equations). Coordinates are always specified via linear polynomials in `Variable`
- All coordinates of rays and lines as well as all coefficients of constraint relations are (arbitrary precision) integers. Only the generators `point()` and `closure_point()` allow one to specify an overall divisor of the otherwise integral coordinates. For example:

```
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0); y = Variable(1)
sage: p = point( 2*x+3*y, 5 ); p
point(2/5, 3/5)
sage: p.coefficient(x)
2
sage: p.coefficient(y)
3
sage: p.divisor()
5
```

- PPL supports (topologically) closed polyhedra (`C_Polyhedron`) as well as not neccesarily closed polyhedra (`NNC_Polyhedron`). Only the latter allows closure points (=points of the closure but not of the actual polyhedron) and strict inequalities (> and <)

The naming convention for the C++ classes is that they start with `PPL_`, for example, the original `Linear_Expression` becomes `PPL_Linear_Expression`. The Python wrapper has the same name as the original library class, that is, just `Linear_Expression`. In short:

- If you are using the Python wrapper (if in doubt: thats you), then you use the same names as the PPL C++ class library.
- If you are writing your own Cython code, you can access the underlying C++ classes by adding the prefix `PPL_` .

Finally, PPL is fast. For example, here is the permutahedron of 5 basis vectors:

```
sage: from sage.libs.ppl import Variable, Generator_System, point, C_Polyhedron
sage: basis = range(0,5)
sage: x = [ Variable(i) for i in basis ]
sage: gs = Generator_System();
sage: for coeff in Permutations(basis):
....:    gs.insert(point( sum( (coeff[i]+1)*x[i] for i in basis ) ))
sage: C_Polyhedron(gs)
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 120 points
```

The above computation (using PPL) finishes without noticeable delay (timeit measures it to be 90 microseconds on sage.math). Below we do the same computation with cddlib, which needs more than 3 seconds on the same hardware:

```
sage: basis = range(0,5)
sage: gs = [ tuple(coeff) for coeff in Permutations(basis) ]
sage: Polyhedron(vertices=gs, backend='cdd')  # long time (3s on sage.math, 2011)
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 120 vertices
```

DIFFERENCES VS. C++

Since Python and C++ syntax are not always compatible, there are necessarily some differences. The main ones are:

- The `Linear_Expression` also accepts an iterable as input for the homogeneous cooefficients.
- `Polyhedron` and its subclasses as well as `Generator_System` and `Constraint_System` can be set immutable via a `set_immutable()` method. This is the analog of declaring a C++ instance `const`. All other classes are immutable by themselves.

AUTHORS:

- Volker Braun (2010-10-08): initial version.
- Risan (2012-02-19): extension for MIP_Problem class

class `sage.libs.ppl.` **`C_Polyhedron`**

Bases: `sage.libs.ppl.Polyhedron`

Wrapper for PPL's `C_Polyhedron` class.

An object of the class `C_Polyhedron` represents a topologically closed convex polyhedron in the vector space. See `NNC_Polyhedron` for more general (not necessarily closed) polyhedra.

When building a closed polyhedron starting from a system of constraints, an exception is thrown if the system contains a strict inequality constraint. Similarly, an exception is thrown when building a closed polyhedron starting from a system of generators containing a closure point.

INPUT:

- `arg` – the defining data of the polyhedron. Any one of the following is accepted:
  - A non-negative integer. Depending on `degenerate_element`, either the space-filling or the empty polytope in the given dimension `arg` is constructed.
  - A `Constraint_System`.
  - A `Generator_System`.
  - A single `Constraint`.
  - A single `Generator`.
  - A `C_Polyhedron`.
- `degenerate_element` – string, either `'universe'` or `'empty'`. Only used if `arg` is an integer.

OUTPUT:

A `C_Polyhedron`.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint, Constraint_System, Generator,␣
↪Generator_System, Variable, C_Polyhedron, point, ray
sage: x = Variable(0)
sage: y = Variable(1)
sage: C_Polyhedron( 5*x-2*y >=  x+y-1 )
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 ray,␣
↪1 line
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: cs.insert( y >= 0 )
sage: C_Polyhedron(cs)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 2 rays
sage: C_Polyhedron( point(x+y) )
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: gs = Generator_System()
```

```
sage: gs.insert( point(-x-y) )
sage: gs.insert( ray(x) )
sage: C_Polyhedron(gs)
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 ray
```

The empty and universe polyhedra are constructed like this:

```
sage: C_Polyhedron(3, 'empty')
The empty polyhedron in QQ^3
sage: C_Polyhedron(3, 'empty').constraints()
Constraint_System {-1==0}
sage: C_Polyhedron(3, 'universe')
The space-filling polyhedron in QQ^3
sage: C_Polyhedron(3, 'universe').constraints()
Constraint_System {}
```

Note that, by convention, the generator system of a polyhedron is either empty or contains at least one point. In particular, if you define a polyhedron via a non-empty *Generator_System* it must contain a point (at any position). If you start with a single generator, this generator must be a point:

```
sage: C_Polyhedron( ray(x) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::C_Polyhedron(gs):
*this is an empty polyhedron and
the non-empty generator system gs contains no points.
```

class sage.libs.ppl.**Constraint**

Bases: object

Wrapper for PPL's Constraint class.

An object of the class Constraint is either:

- an equality $\sum_{i=0}^{n-1} a_i x_i + b = 0$
- a non-strict inequality $\sum_{i=0}^{n-1} a_i x_i + b \geq 0$
- a strict inequality $\sum_{i=0}^{n-1} a_i x_i + b > 0$

where n is the dimension of the space, a_i is the integer coefficient of variable x_i, and b_i is the integer inhomogeneous term.

INPUT/OUTPUT:

You construct constraints by writing inequalities in *Linear_Expression*. Do not attempt to manually construct constraints.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint, Variable, Linear_Expression
sage: x = Variable(0)
sage: y = Variable(1)
sage: 5*x-2*y >  x+y-1
4*x0-3*x1+1>0
sage: 5*x-2*y >= x+y-1
4*x0-3*x1+1>=0
sage: 5*x-2*y == x+y-1
4*x0-3*x1+1==0
sage: 5*x-2*y <= x+y-1
```

```
-4*x0+3*x1-1>=0
sage: 5*x-2*y <  x+y-1
-4*x0+3*x1-1>0
sage: x > 0
x0>0
```

Special care is needed if the left hand side is a constant:

```
sage: 0 == 1    # watch out!
False
sage: Linear_Expression(0) == 1
-1==0
```

OK ( )

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: ineq = (3*x+2*y+1>=0)
sage: ineq.OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'e = (3*x+2*y+1 > 0)\n'
sage: sage_cmd += 'e.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
size 4 1 3 2 -1 > (NNC)
```

coefficient (*v*)

Return the coefficient of the variable `v` .

INPUT:

- `v` – a *Variable* .

OUTPUT:

An integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: ineq = (3*x+1 > 0)
sage: ineq.coefficient(x)
3
```

coefficients ()

Return the coefficients of the constraint.

See also *coefficient()* .

OUTPUT:

A tuple of integers of length *space_dimension()* .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0);  y = Variable(1)
sage: ineq = ( 3*x+5*y+1 ==  2);  ineq
3*x0+5*x1-1==0
sage: ineq.coefficients()
(3, 5)
```

inhomogeneous_term ()

Return the inhomogeneous term of the constraint.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: y = Variable(1)
sage: ineq = ( 10+y > 9 )
sage: ineq
x1+1>0
sage: ineq.inhomogeneous_term()
1
```

is_equality ()

Test whether `self` is an equality.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an equality constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_equality()
True
sage: (x>=0).is_equality()
False
sage: (x>0).is_equality()
False
```

is_equivalent_to (*c*)

Test whether `self` and `c` are equivalent.

INPUT:

- `c` – a *Constraint* .

OUTPUT:

Boolean. Returns `True` if and only if `self` and `c` are equivalent constraints.

Note that constraints having different space dimensions are not equivalent. However, constraints having different types may nonetheless be equivalent, if they both are tautologies or inconsistent.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: x = Variable(0)
sage: y = Variable(1)
sage: ( x>0 ).is_equivalent_to( Linear_Expression(0)<x )
True
sage: ( x>0 ).is_equivalent_to( 0*y<x )
False
sage: ( 0*x>1 ).is_equivalent_to( 0*x==-2 )
True
```

is_inconsistent ( )

Test whether `self` is an inconsistent constraint, that is, always false.

An inconsistent constraint can have either one of the following forms:

- an equality: $\sum 0x_i + b = 0$ with $b \neq 0$,
- a non-strict inequality: $\sum 0x_i + b \geq 0$ with $b < 0$, or
- a strict inequality: $\sum 0x_i + b > 0$ with $b \leq 0$.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an inconsistent constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==1).is_inconsistent()
False
sage: (0*x>=1).is_inconsistent()
True
```

is_inequality ( )

Test whether `self` is an inequality.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an inequality constraint, either strict or non-strict.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_inequality()
False
sage: (x>=0).is_inequality()
True
sage: (x>0).is_inequality()
True
```

is_nonstrict_inequality ( )

Test whether `self` is a non-strict inequality.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an non-strict inequality constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_nonstrict_inequality()
False
sage: (x>=0).is_nonstrict_inequality()
True
sage: (x>0).is_nonstrict_inequality()
False
```

is_strict_inequality ( )

Test whether `self` is a strict inequality.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an strict inequality constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_strict_inequality()
False
sage: (x>=0).is_strict_inequality()
False
sage: (x>0).is_strict_inequality()
True
```

is_tautological ( )

Test whether `self` is a tautological constraint.

A tautology can have either one of the following forms:

- an equality: $\sum 0x_i + 0 = 0$,
- a non-strict inequality: $\sum 0x_i + b \geq 0$ with $b \geq 0$, or
- a strict inequality: $\sum 0x_i + b > 0$ with $b > 0$.

OUTPUT:

Boolean. Returns `True` if and only if `self` is a tautological constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_tautological()
False
sage: (0*x>=0).is_tautological()
True
```

space_dimension ( )

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: (x>=0).space_dimension()
1
sage: (y==1).space_dimension()
2
```

type ( )

Return the constraint type of `self`.

OUTPUT:

String. One of `'equality'`, `'nonstrict_inequality'`, or `'strict_inequality'`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).type()
'equality'
sage: (x>=0).type()
'nonstrict_inequality'
sage: (x>0).type()
'strict_inequality'
```

class `sage.libs.ppl.`**`Constraint_System`**

Bases: `sage.libs.ppl._mutable_or_immutable`

Wrapper for PPL's `Constraint_System` class.

An object of the class Constraint_System is a system of constraints, i.e., a multiset of objects of the class Constraint. When inserting constraints in a system, space dimensions are automatically adjusted so that all the constraints in the system are defined on the same vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint_System, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 5*x-2*y > 0 )
sage: cs.insert( 6*x<3*y )
sage: cs.insert( x >= 2*x-7*y )
sage: cs
Constraint_System {5*x0-2*x1>0, -2*x0+x1>0, -x0+7*x1>=0}
```

OK ( )

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 3*x+2*y+1 <= 10 )
sage: cs.OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Constraint_System, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'cs = Constraint_System( 3*x > 2*y+1 )\n'
sage: sage_cmd += 'cs.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
topology NOT_NECESSARILY_CLOSED
1 x 2 SPARSE (sorted)
index_first_pending 1
size 4 -1 3 -2 -1 > (NNC)
```

clear ( )

Removes all constraints from the constraint system and sets its space dimension to 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System(x>0)
sage: cs
Constraint_System {x0>0}
sage: cs.clear()
sage: cs
Constraint_System {}
```

empty ( )

Return `True` if and only if `self` has no constraints.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, point
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.empty()
True
sage: cs.insert( x>0 )
sage: cs.empty()
False
```

has_equalities ( )

Tests whether `self` contains one or more equality constraints.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
```

```
sage: cs.insert( x<0 )
sage: cs.has_equalities()
False
sage: cs.insert( x==0 )
sage: cs.has_equalities()
True
```

has_strict_inequalities ( )

Tests whether `self` contains one or more strict inequality constraints.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( x==-1 )
sage: cs.has_strict_inequalities()
False
sage: cs.insert( x>0 )
sage: cs.has_strict_inequalities()
True
```

insert (*c*)

Insert `c` into the constraint system.

INPUT:

- `c` – a *`Constraint`*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: cs
Constraint_System {x0>0}
```

space_dimension ( )

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System( x>0 )
sage: cs.space_dimension()
1
```

class `sage.libs.ppl.`**`Constraint_System_iterator`**

Bases: `object`

Wrapper for PPL's `Constraint_System::const_iterator` class.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint_System, Variable, Constraint_System_
↪iterator
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 5*x < 2*y )
sage: cs.insert( 6*x-3*y==0 )
sage: cs.insert( x >= 2*x-7*y )
sage: next(Constraint_System_iterator(cs))
-5*x0+2*x1>0
sage: list(cs)
[-5*x0+2*x1>0, 2*x0-x1==0, -x0+7*x1>=0]
```

next ( )

x.next() -> the next value, or raise StopIteration

class `sage.libs.ppl.`**Generator**

Bases: `object`

Wrapper for PPL's `Generator` class.

An object of the class Generator is one of the following:

- a line $\ell = (a_0, \ldots, a_{n-1})^T$
- a ray $r = (a_0, \ldots, a_{n-1})^T$
- a point $p = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^T$
- a closure point $c = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^T$

where n is the dimension of the space and, for points and closure points, d is the divisor.

INPUT/OUTPUT:

Use the helper functions *`line()`*, *`ray()`*, *`point()`*, and *`closure_point()`* to construct generators. Analogous class methods are also available, see *`Generator.line()`*, *`Generator.ray()`*, *`Generator.point()`*, *`Generator.closure_point()`*. Do not attempt to construct generators manually.

Note: The generators are constructed from linear expressions. The inhomogeneous term is always silently discarded.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: Generator.line(5*x-2*y)
line(5, -2)
sage: Generator.ray(5*x-2*y)
ray(5, -2)
sage: Generator.point(5*x-2*y, 7)
point(5/7, -2/7)
sage: Generator.closure_point(5*x-2*y, 7)
closure_point(5/7, -2/7)
```

OK ( )

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable,␣
↪point\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'p = point(3*x+2*y)\n'
sage: sage_cmd += 'p.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
size 3 1 3 2 P (C)
```

static **closure_point** (*expression=0*, *divisor=1*)

Construct a closure point.

A closure point is a point of the topological closure of a polyhedron that is not a point of the polyhedron itself.

INPUT:

- `expression` – a *Linear_Expression* or something convertible to it (*Variable* or integer).
- `divisor` – an integer.

OUTPUT:

A new *Generator* representing the point.

Raises a `ValueError`` if ``divisor==0`.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.closure_point(2*y+7, 3)
closure_point(0/3, 2/3)
sage: Generator.closure_point(y+7, 3)
closure_point(0/3, 1/3)
sage: Generator.closure_point(7, 3)
closure_point()
sage: Generator.closure_point(0, 0)
Traceback (most recent call last):
...
ValueError: PPL::closure_point(e, d):
d == 0.
```

coefficient (*v*)

Return the coefficient of the variable `v`.

INPUT:

- `v` – a *Variable*.

OUTPUT:

An integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, line
sage: x = Variable(0)
sage: line = line(3*x+1)
sage: line
line(1)
sage: line.coefficient(x)
1
```

coefficients ( )

Return the coefficients of the generator.

See also *coefficient()*.

OUTPUT:

A tuple of integers of length *space_dimension()*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0);  y = Variable(1)
sage: p = point(3*x+5*y+1, 2); p
point(3/2, 5/2)
sage: p.coefficients()
(3, 5)
```

divisor ( )

If `self` is either a point or a closure point, return its divisor.

OUTPUT:

An integer. If `self` is a ray or a line, raises `ValueError`.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: point = Generator.point(2*x-y+5)
sage: point.divisor()
1
sage: line = Generator.line(2*x-y+5)
sage: line.divisor()
Traceback (most recent call last):
...
ValueError: PPL::Generator::divisor():
*this is neither a point nor a closure point.
```

is_closure_point ( )

Test whether `self` is a closure point.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_closure_point()
False
sage: ray(x).is_closure_point()
False
sage: point(x,2).is_closure_point()
False
sage: closure_point(x,2).is_closure_point()
True
```

is_equivalent_to (*g*)

Test whether `self` and `g` are equivalent.

INPUT:

- `g` – a *Generator*.

OUTPUT:

Boolean. Returns `True` if and only if `self` and `g` are equivalent generators.

Note that generators having different space dimensions are not equivalent.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable, point, line
sage: x = Variable(0)
sage: y = Variable(1)
sage: point(2*x    , 2).is_equivalent_to( point(x) )
True
sage: point(2*x+0*y, 2).is_equivalent_to( point(x) )
False
sage: line(4*x).is_equivalent_to(line(x))
True
```

is_line ( )

Test whether `self` is a line.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_line()
True
sage: ray(x).is_line()
False
sage: point(x,2).is_line()
False
sage: closure_point(x,2).is_line()
False
```

is_line_or_ray ( )

Test whether `self` is a line or a ray.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_line_or_ray()
True
sage: ray(x).is_line_or_ray()
True
sage: point(x,2).is_line_or_ray()
False
sage: closure_point(x,2).is_line_or_ray()
False
```

is_point ( )

Test whether `self` is a point.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_point()
False
sage: ray(x).is_point()
False
sage: point(x,2).is_point()
True
sage: closure_point(x,2).is_point()
False
```

is_ray ( )

Test whether `self` is a ray.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_ray()
False
sage: ray(x).is_ray()
True
sage: point(x,2).is_ray()
False
sage: closure_point(x,2).is_ray()
False
```

static **line** (*expression*)

Construct a line.

INPUT:

- `expression` – a *Linear_Expression* or something convertible to it (*Variable* or integer).

OUTPUT:

A new *Generator* representing the line.

Raises a ValueError` if the homogeneous part of ``expression represents the origin of the vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.line(2*y)
line(0, 1)
sage: Generator.line(y)
line(0, 1)
sage: Generator.line(1)
Traceback (most recent call last):
...
ValueError: PPL::line(e):
e == 0, but the origin cannot be a line.
```

static point (*expression=0*, *divisor=1*)

Construct a point.

INPUT:

- `expression` – a *Linear_Expression* or something convertible to it (*Variable* or integer).
- `divisor` – an integer.

OUTPUT:

A new *Generator* representing the point.

Raises a ValueError` if ``divisor==0.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.point(2*y+7, 3)
point(0/3, 2/3)
sage: Generator.point(y+7, 3)
point(0/3, 1/3)
sage: Generator.point(7, 3)
point()
sage: Generator.point(0, 0)
Traceback (most recent call last):
...
ValueError: PPL::point(e, d):
d == 0.
```

static ray (*expression*)

Construct a ray.

INPUT:

- `expression` – a *Linear_Expression* or something convertible to it (*Variable* or integer).

OUTPUT:

A new *Generator* representing the ray.

Raises a `ValueError` if the homogeneous part of ``expression` represents the origin of the vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.ray(2*y)
ray(0, 1)
sage: Generator.ray(y)
ray(0, 1)
sage: Generator.ray(1)
Traceback (most recent call last):
...
ValueError: PPL::ray(e):
e == 0, but the origin cannot be a ray.
```

space_dimension ( )

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: point(x).space_dimension()
1
sage: point(y).space_dimension()
2
```

type ( )

Return the generator type of `self`.

OUTPUT:

String. One of `'line'`, `'ray'`, `'point'`, or `'closure_point'`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).type()
'line'
sage: ray(x).type()
'ray'
sage: point(x,2).type()
'point'
sage: closure_point(x,2).type()
'closure_point'
```

class `sage.libs.ppl.`**Generator_System**

Bases: `sage.libs.ppl._mutable_or_immutable`

Wrapper for PPL's `Generator_System` class.

An object of the class Generator_System is a system of generators, i.e., a multiset of objects of the class Generator (lines, rays, points and closure points). When inserting generators in a system, space dimensions are automatically adjusted so that all the generators in the system are defined on the same vector space. A system of generators which is meant to define a non-empty polyhedron must include at least one point: the reason is that lines, rays and closure points need a supporting point (lines and rays only specify directions while closure points only specify points in the topological closure of the NNC polyhedron).

EXAMPLES:

```
sage: from sage.libs.ppl import Generator_System, Variable, line, ray, point,␣
↪closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( line(5*x-2*y) )
sage: gs.insert( ray(6*x-3*y) )
sage: gs.insert( point(2*x-7*y, 5) )
sage: gs.insert( closure_point(9*x-1*y, 2) )
sage: gs
Generator_System {line(5, -2), ray(2, -1), point(2/5, -7/5), closure_point(9/2, -
↪1/2)}
```

OK ( )

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( point(3*x+2*y+1) )
sage: gs.OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Generator_System, point,␣
↪Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'gs = Generator_System( point(3*x+2*y+1) )\n'
sage: sage_cmd += 'gs.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
topology NECESSARILY_CLOSED
1 x 2 SPARSE (sorted)
index_first_pending 1
size 3 1 3 2 P (C)
```

clear ( )

Removes all generators from the generator system and sets its space dimension to 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) ); gs
Generator_System {point(3/1)}
sage: gs.clear()
sage: gs
Generator_System {}
```

empty ( )

Return `True` if and only if `self` has no generators.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System()
sage: gs.empty()
True
sage: gs.insert( point(-3*x) )
sage: gs.empty()
False
```

insert (*g*)

Insert `g` into the generator system.

The number of space dimensions of `self` is increased, if needed.

INPUT:

- `g` – a *Generator* .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) )
sage: gs.insert( point(-3*x) )
sage: gs
Generator_System {point(3/1), point(-3/1)}
```

space_dimension ( )

Return the dimension of the vector space enclosing `self` .

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) )
sage: gs.space_dimension()
1
```

class `sage.libs.ppl.` **Generator_System_iterator**

Bases: `object`

Wrapper for PPL's `Generator_System::const_iterator` class.

EXAMPLES:

```
sage: from sage.libs.ppl import Generator_System, Variable, line, ray, point,␣
↪closure_point, Generator_System_iterator
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( line(5*x-2*y) )
sage: gs.insert( ray(6*x-3*y) )
sage: gs.insert( point(2*x-7*y, 5) )
sage: gs.insert( closure_point(9*x-1*y, 2) )
sage: next(Generator_System_iterator(gs))
line(5, -2)
sage: list(gs)
[line(5, -2), ray(2, -1), point(2/5, -7/5), closure_point(9/2, -1/2)]
```

next ( )

x.next() -> the next value, or raise StopIteration

class `sage.libs.ppl.`**`Linear_Expression`**

Bases: `object`

Wrapper for PPL's `PPL_Linear_Expression` class.

INPUT:

The constructor accepts zero, one, or two arguments.

If there are two arguments `Linear_Expression(a,b)`, they are interpreted as

- `a` – an iterable of integer coefficients, for example a list.
- `b` – an integer. The inhomogeneous term.

A single argument `Linear_Expression(arg)` is interpreted as

- `arg` – something that determines a linear expression. Possibilities are:
  - a *`Variable`*: The linear expression given by that variable.
  - a *`Linear_Expression`*: The copy constructor.
  - an integer: Constructs the constant linear expression.

No argument is the default constructor and returns the zero linear expression.

OUTPUT:

A *`Linear_Expression`*

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression([1,2,3,4],5)
x0+2*x1+3*x2+4*x3+5
sage: Linear_Expression(10)
10
sage: Linear_Expression()
0
sage: Linear_Expression(10).inhomogeneous_term()
10
sage: x = Variable(123)
sage: expr = x+1; expr
x123+1
```

```
sage: expr.OK()
True
sage: expr.coefficient(x)
1
sage: expr.coefficient( Variable(124) )
0
```

OK ( )

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
```

all_homogeneous_terms_are_zero ( )

Test if `self` is a constant linear expression.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(10).all_homogeneous_terms_are_zero()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'e = 3*x+2*y+1\n'
sage: sage_cmd += 'e.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
size 3 1 3 2
```

coefficient (*v*)

Return the coefficient of the variable `v` .

INPUT:

- `v` – a *Variable* .

OUTPUT:

An integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: e = 3*x+1
sage: e.coefficient(x)
3
```

coefficients ( )

Return the coefficients of the linear expression.

OUTPUT:

A tuple of integers of length *space_dimension()* .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0);  y = Variable(1)
sage: e = 3*x+5*y+1
sage: e.coefficients()
(3, 5)
```

inhomogeneous_term ( )

Return the inhomogeneous term of the linear expression.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(10).inhomogeneous_term()
10
```

is_zero ( )

Test if `self` is the zero linear expression.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(0).is_zero()
True
sage: Linear_Expression(10).is_zero()
False
```

space_dimension ( )

Return the dimension of the vector space necessary for the linear expression.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: ( x+y+1 ).space_dimension()
```

```
2
sage: ( x+y   ).space_dimension()
2
sage: (   y+1 ).space_dimension()
2
sage: ( x   +1 ).space_dimension()
1
sage: ( y+1-y ).space_dimension()
2
```

class sage.libs.ppl.**MIP_Problem**

Bases: sage.libs.ppl._mutable_or_immutable

wrapper for PPL's MIP_Problem class

An object of the class MIP_Problem represents a Mixed Integer (Linear) Program problem.

INPUT:

- dim – integer
- args – an array of the defining data of the MIP_Problem. For each element, any one of the following is accepted:
  - A *Constraint_System*.
  - A *Linear_Expression*.

OUTPUT:

A *MIP_Problem*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.optimal_value()
10/3
sage: m.optimizing_point()
point(10/3, 0/3)
```

OK ( )

Check if all the invariants are satisfied.

OUTPUT:

True if and only if self satisfies all the invariants.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
```

```
sage: m.OK()
True
```

add_constraint (*c*)

Adds a copy of constraint c to the MIP problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.optimal_value()
10/3
```

TESTS:

```
sage: z = Variable(2)
sage: m.add_constraint(z >= -3)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::add_constraint(c):
c.space_dimension() == 3 exceeds this->space_dimension == 2.
```

add_constraints (*cs*)

Adds a copy of the constraints in cs to the MIP problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2)
sage: m.set_objective_function(x + y)
sage: m.add_constraints(cs)
sage: m.optimal_value()
10/3
```

TESTS:

```
sage: p = Variable(9)
sage: cs.insert(p >= -3)
sage: m.add_constraints(cs)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::add_constraints(cs):
cs.space_dimension() == 10 exceeds this->space_dimension() == 2.
```

add_space_dimensions_and_embed (*m*)

Adds m new space dimensions and embeds the old MIP problem in the new vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.add_space_dimensions_and_embed(5)
sage: m.space_dimension()
7
```

add_to_integer_space_dimensions (*i_vars*)

Sets the variables whose indexes are in set i_vars to be integer space dimensions.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set, Constraint_System,␣
↪MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2)
sage: m.set_objective_function(x + y)
sage: m.add_constraints(cs)
sage: i_vars = Variables_Set(x, y)
sage: m.add_to_integer_space_dimensions(i_vars)
sage: m.optimal_value()
3
```

clear ( )

Reset the MIP_Problem to be equal to the trivial MIP_Problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.objective_function()
x0+x1
sage: m.clear()
sage: m.objective_function()
0
```

evaluate_objective_function (*evaluating_point*)

Return the result of evaluating the objective function on evaluating_point. ValueError thrown if self and evaluating_point are dimension-incompatible or if the generator evaluating_point is not a point.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem,␣
↪Generator
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: g = Generator.point(5 * x - 2 * y, 7)
sage: m.evaluate_objective_function(g)
3/7
sage: z = Variable(2)
sage: g = Generator.point(5 * x - 2 * z, 7)
sage: m.evaluate_objective_function(g)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::evaluate_objective_function(p, n, d):
*this and p are dimension incompatible.
```

is_satisfiable ( )

Check if the MIP_Problem is satisfiable

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.is_satisfiable()
True
```

objective_function ( )

Return the optimal value of the MIP_Problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.objective_function()
x0+x1
```

optimal_value ( )

Return the optimal value of the MIP_Problem. ValueError thrown if self does not have an optimizing point, i.e., if the MIP problem is unbounded or not satisfiable.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.optimal_value()
10/3
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: m = MIP_Problem(1, cs, x + x )
sage: m.optimal_value()
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::optimizing_point():
*this does not have an optimizing point.
```

optimization_mode ( )

Return the optimization mode used in the MIP_Problem.

It will return "maximization" if the MIP_Problem was set to MAXIMIZATION mode, and "minimization" otherwise.

EXAMPLES:

```
sage: from sage.libs.ppl import MIP_Problem
sage: m = MIP_Problem()
sage: m.optimization_mode()
'maximization'
```

optimizing_point ( )

Returns an optimal point for the MIP_Problem, if it exists.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.optimizing_point()
point(10/3, 0/3)
```

set_objective_function (*obj*)

Sets the objective function to obj.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
```

```
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.optimal_value()
10/3
```

TESTS:

```
sage: z = Variable(2)
sage: m.set_objective_function(x + y + z)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::set_objective_function(obj):
obj.space_dimension() == 3 exceeds this->space_dimension == 2.
```

set_optimization_mode (*mode*)

Sets the optimization mode to mode.

EXAMPLES:

```
sage: from sage.libs.ppl import MIP_Problem
sage: m = MIP_Problem()
sage: m.optimization_mode()
'maximization'
sage: m.set_optimization_mode('minimization')
sage: m.optimization_mode()
'minimization'
```

TESTS:

```
sage: m.set_optimization_mode('max')
Traceback (most recent call last):
...
ValueError: Unknown value: mode=max.
```

solve ( )

Optimizes the MIP_Problem

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.solve()
{'status': 'optimized'}
```

space_dimension ( )

Return the space dimension of the MIP_Problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.space_dimension()
2
```

class `sage.libs.ppl.`**`NNC_Polyhedron`**

Bases: *sage.libs.ppl.Polyhedron*

Wrapper for PPL's `NNC_Polyhedron` class.

An object of the class `NNC_Polyhedron` represents a not necessarily closed (NNC) convex polyhedron in the vector space.

Note: Since NNC polyhedra are a generalization of closed polyhedra, any object of the class *C_Polyhedron* can be (explicitly) converted into an object of the class *NNC_Polyhedron*. The reason for defining two different classes is that objects of the class *C_Polyhedron* are characterized by a more efficient implementation, requiring less time and memory resources.

INPUT:

- `arg` – the defining data of the polyhedron. Any one of the following is accepted:
  - An non-negative integer. Depending on `degenerate_element`, either the space-filling or the empty polytope in the given dimension `arg` is constructed.
  - A *Constraint_System*.
  - A *Generator_System*.
  - A single *Constraint*.
  - A single *Generator*.
  - A *NNC_Polyhedron*.
  - A *C_Polyhedron*.
- `degenerate_element` – string, either `'universe'` or `'empty'`. Only used if `arg` is an integer.

OUTPUT:

A *C_Polyhedron*.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint, Constraint_System, Generator,
↪Generator_System, Variable, NNC_Polyhedron, point, ray, closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: NNC_Polyhedron( 5*x-2*y >  x+y-1 )
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1
↪closure_point, 1 ray, 1 line
sage: cs = Constraint_System()
sage: cs.insert( x > 0 )
sage: cs.insert( y > 0 )
sage: NNC_Polyhedron(cs)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1
↪closure_point, 2 rays
```

```
sage: NNC_Polyhedron( point(x+y) )
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: gs = Generator_System()
sage: gs.insert( point(-y) )
sage: gs.insert( closure_point(-x-y) )
sage: gs.insert( ray(x) )
sage: p = NNC_Polyhedron(gs); p
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1␣
↪closure_point, 1 ray
sage: p.minimized_constraints()
Constraint_System {x1+1==0, x0+1>0}
```

Note that, by convention, every polyhedron must contain a point:

```
sage: NNC_Polyhedron( closure_point(x+y) )
Traceback (most recent call last):
...
ValueError: PPL::NNC_Polyhedron::NNC_Polyhedron(gs):
*this is an empty polyhedron and
the non-empty generator system gs contains no points.
```

class sage.libs.ppl.**Poly_Con_Relation**

Bases: object

Wrapper for PPL's Poly_Con_Relation class.

INPUT/OUTPUT:

You must not construct *Poly_Con_Relation* objects manually. You will usually get them from *relation_with()* . You can also get pre-defined relations from the class methods *nothing()* , *is_disjoint()* , *strictly_intersects()* , *is_included()* , and *saturates()* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: saturates     = Poly_Con_Relation.saturates();  saturates
saturates
sage: is_included   = Poly_Con_Relation.is_included(); is_included
is_included
sage: is_included.implies(saturates)
False
sage: saturates.implies(is_included)
False
sage: rels = []
sage: rels.append( Poly_Con_Relation.nothing() )
sage: rels.append( Poly_Con_Relation.is_disjoint() )
sage: rels.append( Poly_Con_Relation.strictly_intersects() )
sage: rels.append( Poly_Con_Relation.is_included() )
sage: rels.append( Poly_Con_Relation.saturates() )
sage: rels
[nothing, is_disjoint, strictly_intersects, is_included, saturates]
sage: from sage.matrix.constructor import matrix
sage: m = matrix(5,5)
sage: for i, rel_i in enumerate(rels):
...       for j, rel_j in enumerate(rels):
...           m[i,j] = rel_i.implies(rel_j)
sage: m
[1 0 0 0 0]
[1 1 0 0 0]
```

```
[1 0 1 0 0]
[1 0 0 1 0]
[1 0 0 0 1]
```

OK (*check_non_empty=False*)

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.nothing().OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Poly_Con_Relation\n'
sage: sage_cmd += 'Poly_Con_Relation.nothing().ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
NOTHING
```

implies (*y*)

Test whether `self` implies `y`.

INPUT:

- `y` – a *Poly_Con_Relation*.

OUTPUT:

Boolean. `True` if and only if `self` implies `y`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: nothing = Poly_Con_Relation.nothing()
sage: nothing.implies( nothing )
True
```

static **is_disjoint** ( )

Return the assertion "The polyhedron and the set of points satisfying the constraint are disjoint".

OUTPUT:

A *Poly_Con_Relation*.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_disjoint()
is_disjoint
```

static **is_included** ( )

Return the assertion "The polyhedron is included in the set of points satisfying the constraint".

OUTPUT:

A *Poly_Con_Relation* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_included()
is_included
```

static nothing ( )

Return the assertion that says nothing.

OUTPUT:

A *Poly_Con_Relation* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.nothing()
nothing
```

static saturates ( )

Return the assertion "".

OUTPUT:

A *Poly_Con_Relation* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.saturates()
saturates
```

static strictly_intersects ( )

Return the assertion "The polyhedron intersects the set of points satisfying the constraint, but it is not included in it".

OUTPUT:

A *Poly_Con_Relation* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.strictly_intersects()
strictly_intersects
```

class sage.libs.ppl. **Poly_Gen_Relation**

Bases: `object`

Wrapper for PPL's `Poly_Con_Relation` class.

INPUT/OUTPUT:

You must not construct *Poly_Gen_Relation* objects manually. You will usually get them from *relation_with()* . You can also get pre-defined relations from the class methods *nothing()* and *subsumes()* .

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: nothing = Poly_Gen_Relation.nothing(); nothing
nothing
sage: subsumes = Poly_Gen_Relation.subsumes(); subsumes
subsumes
sage: nothing.implies( subsumes )
False
sage: subsumes.implies( nothing )
True
```

OK (*check_non_empty=False*)

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.nothing().OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Poly_Gen_Relation\n'
sage: sage_cmd += 'Poly_Gen_Relation.nothing().ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd], timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
NOTHING
```

implies (*y*)

Test whether `self` implies `y`.

INPUT:

- `y` – a *Poly_Gen_Relation*.

OUTPUT:

Boolean. `True` if and only if `self` implies `y`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: nothing = Poly_Gen_Relation.nothing()
sage: nothing.implies( nothing )
True
```

static **nothing** ( )

Return the assertion that says nothing.

OUTPUT:

A *Poly_Gen_Relation*.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.nothing()
nothing
```

static subsumes ( )

Return the assertion "Adding the generator would not change the polyhedron".

OUTPUT:

A *Poly_Gen_Relation*.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.subsumes()
subsumes
```

class sage.libs.ppl.**Polyhedron**

Bases: `sage.libs.ppl._mutable_or_immutable`

Wrapper for PPL's `Polyhedron` class.

An object of the class Polyhedron represents a convex polyhedron in the vector space.

A polyhedron can be specified as either a finite system of constraints or a finite system of generators (see Section Representations of Convex Polyhedra) and it is always possible to obtain either representation. That is, if we know the system of constraints, we can obtain from this the system of generators that define the same polyhedron and vice versa. These systems can contain redundant members: in this case we say that they are not in the minimal form.

INPUT/OUTPUT:

This is an abstract base for *C_Polyhedron* and *NNC_Polyhedron*. You cannot instantiate this class.

OK (*check_non_empty=False*)

Check if all the invariants are satisfied.

The check is performed so as to intrude as little as possible. If the library has been compiled with run-time assertions enabled, error messages are written on std::cerr in case invariants are violated. This is useful for the purpose of debugging the library.

INPUT:

- `check_not_empty` – boolean. `True` if and only if, in addition to checking the invariants, `self` must be checked to be not empty.

OUTPUT:

`True` if and only if `self` satisfies all the invariants and either `check_not_empty` is `False` or `self` is not empty.

EXAMPLES:

```
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
```

add_constraint (*c*)

Add a constraint to the polyhedron.

Adds a copy of constraint `c` to the system of constraints of `self`, without minimizing the result.

See alse *add_constraints()*.

INPUT:

- `c` – the *Constraint* that will be added to the system of constraints of `self`.

OUTPUT:

This method modifies the polyhedron `self` and does not return anything.

Raises a `ValueError` if `self` and the constraint `c` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
    sage: from sage.libs.ppl import Variable, C_Polyhedron
    sage: x = Variable(0)
    sage: y = Variable(1)
    sage: p = C_Polyhedron( y>=0 )
    sage: p.add_constraint( x>=0 )

We just added a 1-d constraint to a 2-d polyhedron, this is
fine. The other way is not::

    sage: p = C_Polyhedron( x>=0 )
    sage: p.add_constraint( y>=0 )
    Traceback (most recent call last):
    ...
    ValueError: PPL::C_Polyhedron::add_constraint(c):
    this->space_dimension() == 1, c.space_dimension() == 2.

The constraint must also be topology-compatible, that is,
:class:`C_Polyhedron` only allows non-strict inequalities::

    sage: p = C_Polyhedron( x>=0 )
    sage: p.add_constraint( x< 1 )
    Traceback (most recent call last):
    ...
    ValueError: PPL::C_Polyhedron::add_constraint(c):
    c is a strict inequality.
```

add_constraints (*cs*)

Add constraints to the polyhedron.

Adds a copy of constraints in `cs` to the system of constraints of `self`, without minimizing the result.

See alse *add_constraint()*.

INPUT:

- `cs` – the *Constraint_System* that will be added to the system of constraints of `self`.

OUTPUT:

This method modifies the polyhedron `self` and does not return anything.

Raises a `ValueError` if `self` and the constraints in `cs` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
    sage: from sage.libs.ppl import Variable, C_Polyhedron, Constraint_System
    sage: x = Variable(0)
    sage: y = Variable(1)
    sage: cs = Constraint_System()
    sage: cs.insert(x>=0)
    sage: cs.insert(y>=0)
    sage: p = C_Polyhedron( y<=1 )
    sage: p.add_constraints(cs)

We just added a 1-d constraint to a 2-d polyhedron, this is
fine. The other way is not::

    sage: p = C_Polyhedron( x<=1 )
    sage: p.add_constraints(cs)
    Traceback (most recent call last):
    ...
    ValueError: PPL::C_Polyhedron::add_recycled_constraints(cs):
    this->space_dimension() == 1, cs.space_dimension() == 2.

The constraints must also be topology-compatible, that is,
:class:`C_Polyhedron` only allows non-strict inequalities::

    sage: p = C_Polyhedron( x>=0 )
    sage: p.add_constraints( Constraint_System(x<0) )
    Traceback (most recent call last):
    ...
    ValueError: PPL::C_Polyhedron::add_recycled_constraints(cs):
    cs contains strict inequalities.
```

add_generator (*g*)

Add a generator to the polyhedron.

Adds a copy of constraint `c` to the system of generators of `self`, without minimizing the result.

INPUT:

- `g` – the *Generator* that will be added to the system of Generators of `self`.

OUTPUT:

This method modifies the polyhedron `self` and does not return anything.

Raises a `ValueError` if `self` and the generator `g` are topology-incompatible or dimension-incompatible, or if `self` is an empty polyhedron and `g` is not a point.

EXAMPLES:

```
    sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_
↪point, ray
    sage: x = Variable(0)
    sage: y = Variable(1)
    sage: p = C_Polyhedron(1, 'empty')
    sage: p.add_generator( point(0*x) )

We just added a 1-d generator to a 2-d polyhedron, this is
fine. The other way is not::

    sage: p = C_Polyhedron(1, 'empty')
    sage: p.add_generator(  point(0*y) )
    Traceback (most recent call last):
```

```
   ...
   ValueError: PPL::C_Polyhedron::add_generator(g):
   this->space_dimension() == 1, g.space_dimension() == 2.

The constraint must also be topology-compatible, that is,
:class:`C_Polyhedron` does not allow :func:`closure_point`
generators::

   sage: p = C_Polyhedron( point(0*x+0*y) )
   sage: p.add_generator( closure_point(0*x) )
   Traceback (most recent call last):
   ...
   ValueError: PPL::C_Polyhedron::add_generator(g):
   g is a closure point.
```

Finally, ever non-empty polyhedron must have at least one point generator:

```
sage: p = C_Polyhedron(3, 'empty')
sage: p.add_generator( ray(x) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_generator(g):
*this is an empty polyhedron and g is not a point.
```

add_generators (*gs*)

Add generators to the polyhedron.

Adds a copy of the generators in `gs` to the system of generators of `self`, without minimizing the result.

See alse *add_generator()*.

INPUT:

- `gs` – the *Generator_System* that will be added to the system of constraints of `self`.

OUTPUT:

This method modifies the polyhedron `self` and does not return anything.

Raises a `ValueError` if `self` and one of the generators in `gs` are topology-incompatible or dimension-incompatible, or if `self` is an empty polyhedron and `gs` does not contain a point.

EXAMPLES:

```
   sage: from sage.libs.ppl import Variable, C_Polyhedron, Generator_System,
↪point, ray, closure_point
   sage: x = Variable(0)
   sage: y = Variable(1)
   sage: gs = Generator_System()
   sage: gs.insert(point(0*x+0*y))
   sage: gs.insert(point(1*x+1*y))
   sage: p = C_Polyhedron(2, 'empty')
   sage: p.add_generators(gs)

We just added a 1-d constraint to a 2-d polyhedron, this is
fine. The other way is not::

   sage: p = C_Polyhedron(1, 'empty')
   sage: p.add_generators(gs)
   Traceback (most recent call last):
   ...
```

```
    ValueError: PPL::C_Polyhedron::add_recycled_generators(gs):
    this->space_dimension() == 1, gs.space_dimension() == 2.

The constraints must also be topology-compatible, that is,
:class:`C_Polyhedron` does not allow :func:`closure_point`
generators::

    sage: p = C_Polyhedron( point(0*x+0*y) )
    sage: p.add_generators( Generator_System(closure_point(x) ))
    Traceback (most recent call last):
    ...
    ValueError: PPL::C_Polyhedron::add_recycled_generators(gs):
    gs contains closure points.
```

add_space_dimensions_and_embed (*m*)

Add `m` new space dimensions and embed `self` in the new vector space.

The new space dimensions will be those having the highest indexes in the new polyhedron, which is characterized by a system of constraints in which the variables running through the new dimensions are not constrained. For instance, when starting from the polyhedron P and adding a third space dimension, the result will be the polyhedron

$$\left\{(x, y, z)^T \in \mathbf{R}^3 \middle| (x, y)^T \in P\right\}$$

INPUT:

- `m` – integer.

OUTPUT:

This method assigns the embedded polyhedron to `self` and does not return anything.

Raises a `ValueError` if adding `m` new space dimensions would cause the vector space to exceed dimension `self.max_space_dimension()` .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: p = C_Polyhedron( point(3*x) )
sage: p.add_space_dimensions_and_embed(1)
sage: p.minimized_generators()
Generator_System {line(0, 1), point(3/1, 0/1)}
sage: p.add_space_dimensions_and_embed( p.max_space_dimension() )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_space_dimensions_and_embed(m):
adding m new space dimensions exceeds the maximum allowed space dimension.
```

add_space_dimensions_and_project (*m*)

Add `m` new space dimensions and embed `self` in the new vector space.

The new space dimensions will be those having the highest indexes in the new polyhedron, which is characterized by a system of constraints in which the variables running through the new dimensions are all constrained to be equal to 0. For instance, when starting from the polyhedron P and adding a third space dimension, the result will be the polyhedron

$$\left\{(x, y, 0)^T \in \mathbf{R}^3 \middle| (x, y)^T \in P\right\}$$

INPUT:

- `m` – integer.

OUTPUT:

This method assigns the projected polyhedron to `self` and does not return anything.

Raises a `ValueError` if adding `m` new space dimensions would cause the vector space to exceed dimension `self.max_space_dimension()`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: p = C_Polyhedron( point(3*x) )
sage: p.add_space_dimensions_and_project(1)
sage: p.minimized_generators()
Generator_System {point(3/1, 0/1)}
sage: p.add_space_dimensions_and_project( p.max_space_dimension() )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_space_dimensions_and_project(m):
adding m new space dimensions exceeds the maximum allowed space dimension.
```

affine_dimension ( )

Return the affine dimension of `self`.

OUTPUT:

An integer. Returns 0 if `self` is empty. Otherwise, returns the affine dimension of `self`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( 5*x-2*y == x+y-1 )
sage: p.affine_dimension()
1
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import C_Polyhedron, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'p = C_Polyhedron(3*x+2*y==1)\n'
sage: sage_cmd += 'p.minimized_generators()\n'
sage: sage_cmd += 'p.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time
space_dim 2
-ZE -EM  +CM +GM  +CS +GS  -CP -GP  -SC +SG
con_sys (up-to-date)
topology NECESSARILY_CLOSED
2 x 2 SPARSE (sorted)
index_first_pending 2
size 3 -1 3 2 = (C)
```

```
size 3 1 0 0 >= (C)

gen_sys (up-to-date)
topology NECESSARILY_CLOSED
2 x 2 DENSE (not_sorted)
index_first_pending 2
size 3 0 2 -3 L (C)
size 3 2 0 1 P (C)

sat_c
0 x 0

sat_g
2 x 2
0 0
0 1
```

bounds_from_above (*expr*)

Test whether the `expr` is bounded from above.

INPUT:

- `expr` – a *Linear_Expression*

OUTPUT:

Boolean. Returns `True` if and only if `expr` is bounded from above in `self`.

Raises a `ValueError` if `expr` and `this` are dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
sage: p = C_Polyhedron(y<=0)
sage: p.bounds_from_above(x+1)
False
sage: p.bounds_from_above(Linear_Expression(y))
True
sage: p = C_Polyhedron(x<=0)
sage: p.bounds_from_above(y+1)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::bounds_from_above(e):
this->space_dimension() == 1, e.space_dimension() == 2.
```

bounds_from_below (*expr*)

Test whether the `expr` is bounded from above.

INPUT:

- `expr` – a *Linear_Expression*

OUTPUT:

Boolean. Returns `True` if and only if `expr` is bounded from above in `self`.

Raises a `ValueError` if `expr` and `this` are dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
sage: p = C_Polyhedron(y>=0)
sage: p.bounds_from_below(x+1)
False
sage: p.bounds_from_below(Linear_Expression(y))
True
sage: p = C_Polyhedron(x<=0)
sage: p.bounds_from_below(y+1)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::bounds_from_below(e):
this->space_dimension() == 1, e.space_dimension() == 2.
```

concatenate_assign (*y*)

Assign to `self` the concatenation of `self` and `y`.

This functions returns the Cartiesian product of `self` and `y`.

Viewing a polyhedron as a set of tuples (its points), it is sometimes useful to consider the set of tuples obtained by concatenating an ordered pair of polyhedra. Formally, the concatenation of the polyhedra P and Q (taken in this order) is the polyhedron such that

$$R = \left\{ (x_0, \dots, x_{n-1}, y_0, \dots, y_{m-1})^T \in \mathbf{R}^{n+m} \middle| (x_0, \dots, x_{n-1})^T \in P, \ (y_0, \dots, y_{m-1})^T \in Q \right\}$$

Another way of seeing it is as follows: first embed polyhedron P into a vector space of dimension $n + m$ and then add a suitably renamed-apart version of the constraints defining Q.

INPUT:

- `m` – integer.

OUTPUT:

This method assigns the concatenated polyhedron to `self` and does not return anything.

Raises a `ValueError` if `self` and `y` are topology-incompatible or if adding `y.space_dimension()` new space dimensions would cause the vector space to exceed dimension `self.max_space_dimension()`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron, point
sage: x = Variable(0)
sage: p1 = C_Polyhedron( point(1*x) )
sage: p2 = C_Polyhedron( point(2*x) )
sage: p1.concatenate_assign(p2)
sage: p1.minimized_generators()
Generator_System {point(1/1, 2/1)}
```

The polyhedra must be topology-compatible and not exceed the maximum space dimension:

```
sage: p1.concatenate_assign( NNC_Polyhedron(1, 'universe') )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::concatenate_assign(y):
y is a NNC_Polyhedron.
sage: p1.concatenate_assign( C_Polyhedron(p1.max_space_dimension(), 'empty') )
Traceback (most recent call last):
...
```

```
ValueError: PPL::C_Polyhedron::concatenate_assign(y):
concatenation exceeds the maximum allowed space dimension.
```

constrains (*var*)

Test whether `var` is constrained in `self`.

INPUT:

- `var` – a *Variable*.

OUTPUT:

Boolean. Returns `True` if and only if `var` is constrained in `self`.

Raises a `ValueError` if `var` is not a space dimension of `self`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: p = C_Polyhedron(1, 'universe')
sage: p.constrains(x)
False
sage: p = C_Polyhedron(x>=0)
sage: p.constrains(x)
True
sage: y = Variable(1)
sage: p.constrains(y)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::constrains(v):
this->space_dimension() == 1, v.space_dimension() == 2.
```

constraints ( )

Returns the system of constraints.

See also *minimized_constraints()*.

OUTPUT:

A *Constraint_System*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( y>=0 )
sage: p.add_constraint( x>=0 )
sage: p.add_constraint( x+y>=0 )
sage: p.constraints()
Constraint_System {x1>=0, x0>=0, x0+x1>=0}
sage: p.minimized_constraints()
Constraint_System {x1>=0, x0>=0}
```

contains (*y*)

Test whether `self` contains `y`.

INPUT:

- `y` – a *Polyhedron*.

OUTPUT:

Boolean. Returns `True` if and only if `self` contains `y`.

Raises a `ValueError` if `self` and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p0 = C_Polyhedron( x>=0 )
sage: p1 = C_Polyhedron( x>=1 )
sage: p0.contains(p1)
True
sage: p1.contains(p0)
False
```

Errors are raised if the dimension or topology is not compatible:

```
sage: p0.contains(C_Polyhedron(y>=0))
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::contains(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p0.contains(NNC_Polyhedron(x>0))
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::contains(y):
y is a NNC_Polyhedron.
```

contains_integer_point ()

Test whether `self` contains an integer point.

OUTPUT:

Boolean. Returns `True` if and only if `self` contains an integer point.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron(x>0)
sage: p.add_constraint(x<1)
sage: p.contains_integer_point()
False
sage: p.topological_closure_assign()
sage: p.contains_integer_point()
True
```

difference_assign (*y*)

Assign to `self` the poly-difference of `self` and `y`.

For any pair of NNC polyhedra P_1 and P_2 the convex polyhedral difference (or poly-difference) of P_1 and P_2 is defined as the smallest convex polyhedron containing the set-theoretic difference $P_1 \setminus P_2$ of P_1 and P_2.

In general, even if P_1 and P_2 are topologically closed polyhedra, their poly-difference may be a convex polyhedron that is not topologically closed. For this reason, when computing the poly-difference of two *C_Polyhedron*, the library will enforce the topological closure of the result.

INPUT:

- y – a *Polyhedron*

OUTPUT:

This method assigns the poly-difference to `self` and does not return anything.

Raises a `ValueError` if `self` and and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_point,␣
↪NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(NNC_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {-x0+1>=0, x0>0}
```

The poly-difference of `C_polyhedron` is really its closure:

```
sage: p = C_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(C_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {x0>=0, -x0+1>=0}
```

`self` and `y` must be dimension- and topology-compatible, or an exception is raised:

```
sage: y = Variable(1)
sage: p.poly_difference_assign( C_Polyhedron(y>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p.poly_difference_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
y is a NNC_Polyhedron.
```

drop_some_non_integer_points ( )

Possibly tighten `self` by dropping some points with non-integer coordinates.

The modified polyhedron satisfies:

- it is (not necessarily strictly) contained in the original polyhedron.
- integral vertices (generating points with integer coordinates) of the original polyhedron are not removed.

Note: The modified polyhedron is not neccessarily a lattice polyhedron; Some vertices will, in general, still be rational. Lattice points interior to the polyhedron may be lost in the process.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, NNC_Polyhedron, Constraint_System
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+2*y<5 )
sage: p = NNC_Polyhedron(cs)
sage: p.minimized_generators()
Generator_System {point(0/1, 0/1), closure_point(0/2, 5/2), closure_point(5/
↪3, 0/3)}
sage: p.drop_some_non_integer_points()
sage: p.minimized_generators()
Generator_System {point(0/1, 0/1), point(0/1, 2/1), point(4/3, 0/3)}
```

generators ( )

Returns the system of generators.

See also *minimized_generators()* .

OUTPUT:

A *Generator_System* .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3,'empty')
sage: p.add_generator( point(-x-y) )
sage: p.add_generator( point(0) )
sage: p.add_generator( point(+x+y) )
sage: p.generators()
Generator_System {point(-1/1, -1/1, 0/1), point(0/1, 0/1, 0/1), point(1/1, 1/
↪1, 0/1)}
sage: p.minimized_generators()
Generator_System {point(-1/1, -1/1, 0/1), point(1/1, 1/1, 0/1)}
```

intersection_assign (*y*)

Assign to `self` the intersection of `self` and `y` .

INPUT:

- `y` – a *Polyhedron*

OUTPUT:

This method assigns the intersection to `self` and does not return anything.

Raises a `ValueError` if `self` and and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( 1*x+0*y >= 0 )
sage: p.intersection_assign( C_Polyhedron(y>=0) )
sage: p.constraints()
Constraint_System {x0>=0, x1>=0}
```

```
sage: z = Variable(2)
sage: p.intersection_assign( C_Polyhedron(z>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::intersection_assign(y):
this->space_dimension() == 2, y.space_dimension() == 3.
sage: p.intersection_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::intersection_assign(y):
y is a NNC_Polyhedron.
```

is_bounded ( )

Test whether `self` is bounded.

OUTPUT:

Boolean. Returns `True` if and only if `self` is a bounded polyhedron.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, NNC_Polyhedron, point, closure_
↪point, ray
sage: x = Variable(0)
sage: p = NNC_Polyhedron( point(0*x) )
sage: p.add_generator( closure_point(1*x) )
sage: p.is_bounded()
True
sage: p.add_generator( ray(1*x) )
sage: p.is_bounded()
False
```

is_discrete ( )

Test whether `self` is discrete.

OUTPUT:

Boolean. Returns `True` if and only if `self` is discrete.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, ray
sage: x = Variable(0); y = Variable(1)
sage: p = C_Polyhedron( point(1*x+2*y) )
sage: p.is_discrete()
True
sage: p.add_generator( point(x) )
sage: p.is_discrete()
False
```

is_disjoint_from (*y*)

Tests whether `self` and `y` are disjoint.

INPUT:

- `y` – a *Polyhedron*.

OUTPUT:

Boolean. Returns `True` if and only if `self` and `y` are disjoint.

Rayises a `ValueError` if `self` and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0);   y = Variable(1)
sage: C_Polyhedron(x<=0).is_disjoint_from( C_Polyhedron(x>=1) )
True
```

This is not allowed:

```
sage: x = Variable(0);   y = Variable(1)
sage: poly_1d = C_Polyhedron(x<=0)
sage: poly_2d = C_Polyhedron(x+0*y>=1)
sage: poly_1d.is_disjoint_from(poly_2d)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::intersection_assign(y):
this->space_dimension() == 1, y.space_dimension() == 2.
```

Nor is this:

```
sage: x = Variable(0);   y = Variable(1)
sage: c_poly   =   C_Polyhedron( x<=0 )
sage: nnc_poly = NNC_Polyhedron( x >0 )
sage: c_poly.is_disjoint_from(nnc_poly)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::intersection_assign(y):
y is a NNC_Polyhedron.
sage: NNC_Polyhedron(c_poly).is_disjoint_from(nnc_poly)
True
```

is_empty ()

Test if `self` is an empty polyhedron.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(3, 'empty').is_empty()
True
sage: C_Polyhedron(3, 'universe').is_empty()
False
```

is_topologically_closed ()

Tests if `self` is topologically closed.

OUTPUT:

Returns `True` if and only if `self` is a topologically closed subset of the ambient vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0);   y = Variable(1)
sage: C_Polyhedron(3, 'universe').is_topologically_closed()
True
sage: C_Polyhedron( x>=1 ).is_topologically_closed()
```

```
True
sage: NNC_Polyhedron( x>1 ).is_topologically_closed()
False
```

is_universe ( )

Test if `self` is a universe (space-filling) polyhedron.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(3, 'empty').is_universe()
False
sage: C_Polyhedron(3, 'universe').is_universe()
True
```

max_space_dimension ( )

Return the maximum space dimension all kinds of Polyhedron can handle.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(1, 'empty').max_space_dimension()   # random output
1152921504606846974
sage: C_Polyhedron(1, 'empty').max_space_dimension()
357913940            # 32-bit
1152921504606846974  # 64-bit
```

maximize (*expr*)

Maximize `expr` .

INPUT:

- `expr` – a *Linear_Expression* .

OUTPUT:

A dictionary with the following keyword:value pair:

- `'bounded'` : Boolean. Whether the linear expression `expr` is bounded from above on `self` .

If `expr` is bounded from above, the following additional keyword:value pairs are set to provide information about the supremum:

- `'sup_n'` : Integer. The numerator of the supremum value.
- `'sup_d'` : Non-zero integer. The denominator of the supremum value.
- `'maximum'` : Boolean. `True` if and only if the supremum is also the maximum value.
- `'generator'` : a *Generator* . A point or closure point where expr reaches its supremum value.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron,
↪Constraint_System, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
```

```
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+5*y<=10 )
sage: p = C_Polyhedron(cs)
sage: p.maximize( x+y )
{'bounded': True,
 'generator': point(10/3, 0/3),
 'maximum': True,
 'sup_d': 3,
 'sup_n': 10}
```

Unbounded case:

```
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: p = NNC_Polyhedron(cs)
sage: p.maximize( +x )
{'bounded': False}
sage: p.maximize( -x )
{'bounded': True,
 'generator': closure_point(0/1),
 'maximum': False,
 'sup_d': 1,
 'sup_n': 0}
```

minimize (*expr*)

Minimize `expr` .

INPUT:

- `expr` – a *Linear_Expression* .

OUTPUT:

A dictionary with the following keyword:value pair:

- `'bounded'` : Boolean. Whether the linear expression `expr` is bounded from below on `self` .

If `expr` is bounded from below, the following additional keyword:value pairs are set to provide information about the infimum:

- `'inf_n'` : Integer. The numerator of the infimum value.
- `'inf_d'` : Non-zero integer. The denominator of the infimum value.
- `'minimum'` : Boolean. `True` if and only if the infimum is also the minimum value.
- `'generator'` : a *Generator* . A point or closure point where expr reaches its infimum value.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron,␣
↪Constraint_System, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+5*y<=10 )
sage: p = C_Polyhedron(cs)
sage: p.minimize( x+y )
{'bounded': True,
```

```
 'generator': point(0/1, 0/1),
 'inf_d': 1,
 'inf_n': 0,
 'minimum': True}
```

Unbounded case:

```
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: p = NNC_Polyhedron(cs)
sage: p.minimize( +x )
{'bounded': True,
 'generator': closure_point(0/1),
 'inf_d': 1,
 'inf_n': 0,
 'minimum': False}
sage: p.minimize( -x )
{'bounded': False}
```

minimized_constraints ( )

Returns the minimized system of constraints.

See also *constraints()* .

OUTPUT:

A *Constraint_System*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( y>=0 )
sage: p.add_constraint( x>=0 )
sage: p.add_constraint( x+y>=0 )
sage: p.constraints()
Constraint_System {x1>=0, x0>=0, x0+x1>=0}
sage: p.minimized_constraints()
Constraint_System {x1>=0, x0>=0}
```

minimized_generators ( )

Returns the minimized system of generators.

See also *generators()* .

OUTPUT:

A *Generator_System*.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3,'empty')
sage: p.add_generator( point(-x-y) )
sage: p.add_generator( point(0) )
sage: p.add_generator( point(+x+y) )
sage: p.generators()
Generator_System {point(-1/1, -1/1, 0/1), point(0/1, 0/1, 0/1), point(1/1, 1/
↪1, 0/1)}
```

```
sage: p.minimized_generators()
Generator_System {point(-1/1, -1/1, 0/1), point(1/1, 1/1, 0/1)}
```

poly_difference_assign (*y*)

Assign to `self` the poly-difference of `self` and `y`.

For any pair of NNC polyhedra P_1 and P_2 the convex polyhedral difference (or poly-difference) of P_1 and P_2 is defined as the smallest convex polyhedron containing the set-theoretic difference $P_1 \setminus P_2$ of P_1 and P_2.

In general, even if P_1 and P_2 are topologically closed polyhedra, their poly-difference may be a convex polyhedron that is not topologically closed. For this reason, when computing the poly-difference of two *C_Polyhedron*, the library will enforce the topological closure of the result.

INPUT:

- `y` – a *Polyhedron*

OUTPUT:

This method assigns the poly-difference to `self` and does not return anything.

Raises a `ValueError` if `self` and and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_point,␣
↪NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(NNC_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {-x0+1>=0, x0>0}
```

The poly-difference of `C_polyhedron` is really its closure:

```
sage: p = C_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(C_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {x0>=0, -x0+1>=0}
```

`self` and `y` must be dimension- and topology-compatible, or an exception is raised:

```
sage: y = Variable(1)
sage: p.poly_difference_assign( C_Polyhedron(y>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p.poly_difference_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
y is a NNC_Polyhedron.
```

poly_hull_assign (*y*)

Assign to `self` the poly-hull of `self` and `y`.

For any pair of NNC polyhedra P_1 and P_2, the convex polyhedral hull (or poly-hull) of is the smallest NNC polyhedron that includes both P_1 and P_2. The poly-hull of any pair of closed polyhedra in is also closed.

INPUT:

- `y` – a *Polyhedron*

OUTPUT:

This method assigns the poly-hull to `self` and does not return anything.

Raises a `ValueError` if `self` and and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(1*x+0*y) )
sage: p.poly_hull_assign(C_Polyhedron( point(0*x+1*y) ))
sage: p.generators()
Generator_System {point(0/1, 1/1), point(1/1, 0/1)}
```

`self` and `y` must be dimension- and topology-compatible, or an exception is raised:

```
sage: z = Variable(2)
sage: p.poly_hull_assign( C_Polyhedron(z>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
this->space_dimension() == 2, y.space_dimension() == 3.
sage: p.poly_hull_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
y is a NNC_Polyhedron.
```

relation_with (*arg*)

Return the relations holding between the polyhedron `self` and the generator or constraint `arg`.

INPUT:

- `arg` – a *Generator* or a *Constraint*.

OUTPUT:

A *Poly_Gen_Relation* or a *Poly_Con_Relation* according to the type of the input.

Raises `ValueError` if `self` and the generator/constraint `arg` are dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, ray, Poly_Con_
↪Relation
sage: x = Variable(0);  y = Variable(1)
sage: p = C_Polyhedron(2, 'empty')
sage: p.add_generator( point(1*x+0*y) )
sage: p.add_generator( point(0*x+1*y) )
sage: p.minimized_constraints()
Constraint_System {x0+x1-1==0, -x1+1>=0, x1>=0}
sage: p.relation_with( point(1*x+1*y) )
nothing
```

```
sage: p.relation_with( point(1*x+1*y, 2) )
subsumes
sage: p.relation_with( x+y==-1 )
is_disjoint
sage: p.relation_with( x==y )
strictly_intersects
sage: p.relation_with( x+y<=1 )
is_included, saturates
sage: p.relation_with( x+y<1 )
is_disjoint, saturates
```

In a Sage program you will usually use *relation_with()* together with *implies()* or *implies()*, for example:

```
sage: p.relation_with( x+y<1 ).implies(Poly_Con_Relation.saturates())
True
```

You can only get relations with dimension-compatible generators or constraints:

```
sage: z = Variable(2)
sage: p.relation_with( point(x+y+z) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::relation_with(g):
this->space_dimension() == 2, g.space_dimension() == 3.
sage: p.relation_with( z>0 )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::relation_with(c):
this->space_dimension() == 2, c.space_dimension() == 3.
```

remove_higher_space_dimensions (*new_dimension*)

Remove the higher dimensions of the vector space so that the resulting space will have dimension `new_dimension`.

OUTPUT:

This method modifies `self` and does not return anything.

Raises a `ValueError` if `new_dimensions` is greater than the space dimension of `self`.

EXAMPLES:

```
sage: from sage.libs.ppl import C_Polyhedron, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3*x+0*y==2)
sage: p.remove_higher_space_dimensions(1)
sage: p.minimized_constraints()
Constraint_System {3*x0-2==0}
sage: p.remove_higher_space_dimensions(2)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::remove_higher_space_dimensions(nd):
this->space_dimension() == 1, required space dimension == 2.
```

space_dimension ( )

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( 5*x-2*y >=  x+y-1 )
sage: p.space_dimension()
2
```

strictly_contains (*y*)

Test whether `self` strictly contains `y`.

INPUT:

- `y` – a *Polyhedron*.

OUTPUT:

Boolean. Returns `True` if and only if `self` contains `y` and `self` does not equal `y`.

Raises a `ValueError` if `self` and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p0 = C_Polyhedron( x>=0 )
sage: p1 = C_Polyhedron( x>=1 )
sage: p0.strictly_contains(p1)
True
sage: p1.strictly_contains(p0)
False
```

Errors are raised if the dimension or topology is not compatible:

```
sage: p0.strictly_contains(C_Polyhedron(y>=0))
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::contains(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p0.strictly_contains(NNC_Polyhedron(x>0))
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::contains(y):
y is a NNC_Polyhedron.
```

topological_closure_assign ( )

Assign to `self` its topological closure.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron(x>0)
sage: p.is_topologically_closed()
False
sage: p.topological_closure_assign()
```

```
sage: p.is_topologically_closed()
True
sage: p.minimized_constraints()
Constraint_System {x0>=0}
```

unconstrain (*var*)

Compute the cylindrification of `self` with respect to space dimension `var` .

INPUT:

- `var` – a *Variable* . The space dimension that will be unconstrained. Exceptions:

OUTPUT:

This method assigns the cylindrification to `self` and does not return anything.

Raises a `ValueError` if `var` is not a space dimension of `self` .

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(x+y) ); p
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: p.unconstrain(x); p
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1␣
↪line
sage: z = Variable(2)
sage: p.unconstrain(z)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::unconstrain(var):
this->space_dimension() == 2, required space dimension == 3.
```

upper_bound_assign (*y*)

Assign to `self` the poly-hull of `self` and `y` .

For any pair of NNC polyhedra P_1 and P_2, the convex polyhedral hull (or poly-hull) of is the smallest NNC polyhedron that includes both P_1 and P_2. The poly-hull of any pair of closed polyhedra in is also closed.

INPUT:

- `y` – a *Polyhedron*

OUTPUT:

This method assigns the poly-hull to `self` and does not return anything.

Raises a `ValueError` if `self` and and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(1*x+0*y) )
sage: p.poly_hull_assign(C_Polyhedron( point(0*x+1*y) ))
sage: p.generators()
Generator_System {point(0/1, 1/1), point(1/1, 0/1)}
```

`self` and `y` must be dimension- and topology-compatible, or an exception is raised:

```
sage: z = Variable(2)
sage: p.poly_hull_assign( C_Polyhedron(z>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
this->space_dimension() == 2, y.space_dimension() == 3.
sage: p.poly_hull_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
y is a NNC_Polyhedron.
```

class `sage.libs.ppl.`**`Variable`**

Bases: `object`

Wrapper for PPL's `Variable` class.

A dimension of the vector space.

An object of the class Variable represents a dimension of the space, that is one of the Cartesian axes. Variables are used as basic blocks in order to build more complex linear expressions. Each variable is identified by a non-negative integer, representing the index of the corresponding Cartesian axis (the first axis has index 0). The space dimension of a variable is the dimension of the vector space made by all the Cartesian axes having an index less than or equal to that of the considered variable; thus, if a variable has index i, its space dimension is $i + 1$.

INPUT:

- `i` – integer. The index of the axis.

OUTPUT:

A *`Variable`*

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(123)
sage: x.id()
123
sage: x
x123
```

Note that the "meaning" of an object of the class Variable is completely specified by the integer index provided to its constructor: be careful not to be mislead by C++ language variable names. For instance, in the following example the linear expressions `e1` and `e2` are equivalent, since the two variables `x` and `z` denote the same Cartesian axis:

```
sage: x = Variable(0)
sage: y = Variable(1)
sage: z = Variable(0)
sage: e1 = x + y; e1
x0+x1
sage: e2 = y + z; e2
x0+x1
sage: e1 - e2
0
```

OK ( )

Checks if all the invariants are satisfied.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: x.OK()
True
```

id ( )

Return the index of the Cartesian axis associated to the variable.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(123)
sage: x.id()
123
```

space_dimension ( )

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer. The returned value is `self.id()+1`.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: x.space_dimension()
1
```

class `sage.libs.ppl.`**Variables_Set**

Bases: `object`

Wrapper for PPL's `Variables_Set` class.

A set of variables' indexes.

EXAMPLES:

Build the empty set of variable indexes:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: Variables_Set()
Variables_Set of cardinality 0
```

Build the singleton set of indexes containing the index of the variable:

```
sage: v123 = Variable(123)
sage: Variables_Set(v123)
Variables_Set of cardinality 1
```

Build the set of variables' indexes in the range from one variable to another variable:

```
sage: v127 = Variable(127)
sage: Variables_Set(v123,v127)
Variables_Set of cardinality 5
```

OK ( )

Checks if all the invariants are satisfied.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: v123 = Variable(123)
sage: S = Variables_Set(v123)
sage: S.OK()
True
```

ascii_dump ( )

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Variable, Variables_Set\n'
sage: sage_cmd += 'v123 = Variable(123)\n'
sage: sage_cmd += 'S = Variables_Set(v123)\n'
sage: sage_cmd += 'S.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],␣
↪timeout=100)  # long time, indirect doctest
sage: print(err)  # long time

variables( 1 )
123
```

insert (v)

Inserts the index of variable v into the set.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: S = Variables_Set()
sage: v123 = Variable(123)
sage: S.insert(v123)
sage: S.space_dimension()
124
```

space_dimension ( )

Returns the dimension of the smallest vector space enclosing all the variables whose indexes are in the set.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: v123 = Variable(123)
sage: S = Variables_Set(v123)
```

```
sage: S.space_dimension()
124
```

sage.libs.ppl.**closure_point**(*expression=0*, *divisor=1*)

Constuct a closure point.

See `Generator.closure_point()` for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, closure_point
sage: y = Variable(1)
sage: closure_point(2*y, 5)
closure_point(0/5, 2/5)
```

sage.libs.ppl.**equation**(*expression*)

Constuct an equation.

INPUT:

- `expression` – a `Linear_Expression`.

OUTPUT:

The equation `expression` == 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, equation
sage: y = Variable(1)
sage: 2*y+1 == 0
2*x1+1==0
sage: equation(2*y+1)
2*x1+1==0
```

sage.libs.ppl.**inequality**(*expression*)

Constuct an inequality.

INPUT:

- `expression` – a `Linear_Expression`.

OUTPUT:

The inequality `expression` >= 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, inequality
sage: y = Variable(1)
sage: 2*y+1 >= 0
2*x1+1>=0
sage: inequality(2*y+1)
2*x1+1>=0
```

sage.libs.ppl.**line**(*expression*)

Constuct a line.

See `Generator.line()` for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, line
sage: y = Variable(1)
sage: line(2*y)
line(0, 1)
```

sage.libs.ppl.**point**(*expression=0, divisor=1*)

Constuct a point.

See *Generator.point()* for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, point
sage: y = Variable(1)
sage: point(2*y, 5)
point(0/5, 2/5)
```

sage.libs.ppl.**ray**(*expression*)

Constuct a ray.

See *Generator.ray()* for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, ray
sage: y = Variable(1)
sage: ray(2*y)
ray(0, 1)
```

sage.libs.ppl.**strict_inequality**(*expression*)

Constuct a strict inequality.

INPUT:

- expression – a *Linear_Expression*.

OUTPUT:

The inequality `expression` > 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, strict_inequality
sage: y = Variable(1)
sage: 2*y+1 > 0
2*x1+1>0
sage: strict_inequality(2*y+1)
2*x1+1>0
```

CHAPTER

EIGHTEEN

LINBOX INTERFACE

class `sage.libs.linbox.linbox.`**`Linbox_integer_dense`**

Bases: `object`

`charpoly``()`

OUTPUT:

coefficients of charpoly or minpoly as a Python list

`det``()`

OUTPUT:

determinant as a sage Integer

`minpoly``()`

OUTPUT:

coefficients of minpoly as a Python list

`smithform``()`

class `sage.libs.linbox.linbox.`**`Linbox_modn_sparse`**

Bases: `object`

CHAPTER

NINETEEN

FLINT IMPORTS

TESTS:

Import this module:

```
sage: import sage.libs.flint.flint
```

We verify that trac ticket #6919 is correctly fixed:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: A = 2^(2^17+2^15)
sage: a = A * x^31
sage: b = (A * x) * x^30
sage: a == b
True
```

`sage.libs.flint.flint.` **`free_flint_stack`** `()`

CHAPTER

TWENTY

FLINT FMPZ_POLY CLASS WRAPPER

AUTHORS:

- Robert Bradshaw (2007-09-15) Initial version.
- William Stein (2007-10-02) update for new flint; add arithmetic and creation of coefficients of arbitrary size.

class sage.libs.flint.fmpz_poly.**Fmpz_poly**

Bases: sage.structure.sage_object.SageObject

Construct a new fmpz_poly from a sequence, constant coefficient, or string (in the same format as it prints).

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: Fmpz_poly([1,2,3])
3  1 2 3
sage: Fmpz_poly(5)
1  5
sage: Fmpz_poly(str(Fmpz_poly([3,5,7])))
3  3 5 7
```

degree()

The degree of self.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2,3]); f
3  1 2 3
sage: f.degree()
2
sage: Fmpz_poly(range(1000)).degree()
999
sage: Fmpz_poly([2,0]).degree()
0
```

derivative()

Return the derivative of self.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2,6])
sage: f.derivative().list() == [2, 12]
True
```

div_rem (*other*)
Return self / other, self, % other.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,3,4,5])
sage: g = f^23
sage: g.div_rem(f)[1]
0
sage: g.div_rem(f)[0] - f^22
0
sage: f = Fmpz_poly([1..10])
sage: g = Fmpz_poly([1,3,5])
sage: q, r = f.div_rem(g)
sage: q*f+r
17  1 2 3 4 4 4 10 11 17 18 22 26 30 23 26 18 20
sage: g
3  1 3 5
sage: q*g+r
10  1 2 3 4 5 6 7 8 9 10
```

left_shift (*n*)
Left shift self by n.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2])
sage: f.left_shift(1).list() == [0,1,2]
True
```

list ( )
Return self as a list of coefficients, lowest terms first.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([2,1,0,-1])
sage: f.list()
[2, 1, 0, -1]
```

pow_truncate (*exp*, *n*)
Return self raised to the power of exp mod x^n.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2])
sage: f.pow_truncate(10,3)
3  1 20 180
sage: f.pow_truncate(1000,3)
3  1 2000 1998000
```

pseudo_div (*other*)

pseudo_div_rem (*other*)

right_shift (*n*)
Right shift self by n.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2])
sage: f.right_shift(1).list() == [2]
True
```

truncate (*n*)

Return the truncation of self at degree n.

EXAMPLES:

```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,1])
sage: g = f**10; g
11  1 10 45 120 210 252 210 120 45 10 1
sage: g.truncate(5)
5  1 10 45 120 210
```

CHAPTER

TWENTYONE

FLINT ARITHMETIC FUNCTIONS

sage.libs.flint.arith.**bell_number**(*n*)

Returns the n th Bell number.

EXAMPLES:

```
sage: from sage.libs.flint.arith import bell_number
sage: [bell_number(i) for i in range(10)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
sage: bell_number(10)
115975
sage: bell_number(40)
157450588391204931289324344702531067
sage: bell_number(100)
47585391276764833658790768841387207826363669686825611466616334637559114497892442622672724044217
```

sage.libs.flint.arith.**bernoulli_number**(*n*)

Return the n-th Bernoulli number.

EXAMPLES:

```
sage: from sage.libs.flint.arith import bernoulli_number
sage: [bernoulli_number(i) for i in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
sage: bernoulli_number(10)
5/66
sage: bernoulli_number(40)
-261082718496449122051/13530
sage: bernoulli_number(100)
-
↪94598037819122125295227433069493721872702841533066936133385696204311395415197247711/
↪33330
```

sage.libs.flint.arith.**dedekind_sum**(*p*, *q*)

Return the Dedekind sum $s(p, q)$ where p and q are arbitrary integers.

EXAMPLES:

```
sage: from sage.libs.flint.arith import dedekind_sum
sage: dedekind_sum(4, 5)
-1/5
```

sage.libs.flint.arith.**euler_number**(*n*)

Return the Euler number of index n.

EXAMPLES:

```
sage: from sage.libs.flint.arith import euler_number
sage: [euler_number(i) for i in range(8)]
[1, 0, -1, 0, 5, 0, -61, 0]
```

sage.libs.flint.arith.**harmonic_number**(*n*)

Returns the harmonic number `H_n`.

EXAMPLES:

```
sage: from sage.libs.flint.arith import harmonic_number
sage: n = 500 + randint(0,500)
sage: bool( sum(1/k for k in range(1,n+1)) == harmonic_number(n) )
True
```

sage.libs.flint.arith.**number_of_partitions**(*n*)

Returns the number of partitions of the integer `n`.

EXAMPLES:

```
sage: from sage.libs.flint.arith import number_of_partitions
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
37338
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055050741660
```

TESTS:

```
sage: n = 500 + randint(0,500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1500 + randint(0,1500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 100000000 + randint(0,100000000)
```

```
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0  # long time
True
```

CHAPTER

TWENTYTWO

SYMMETRICA LIBRARY

sage.libs.symmetrica.symmetrica.**bdg_symmetrica**(*part*, *perm*)

Calculates the irreduzible matrix representation D^part(perm), whose entries are of integral numbers.

REFERENCE: H. Boerner: Darstellungen von Gruppen, Springer 1955. pp. 104-107.

sage.libs.symmetrica.symmetrica.**chartafel_symmetrica**(*n*)

you enter the degree of the symmetric group, as INTEGER object and the result is a MATRIX object: the charactertable of the symmetric group of the given degree.

EXAMPLES:

```
sage: symmetrica.chartafel(3)
[ 1  1  1]
[-1  0  2]
[ 1 -1  1]
sage: symmetrica.chartafel(4)
[ 1  1  1  1  1]
[-1  0 -1  1  3]
[ 0 -1  2  0  2]
[ 1  0 -1 -1  3]
[-1  1  1 -1  1]
```

sage.libs.symmetrica.symmetrica.**charvalue_symmetrica**(*irred*, *cls*, *table=None*)

you enter a PARTITION object part, labelling the irreducible character, you enter a PARTITION object class, labeling the class or class may be a PERMUTATION object, then result becomes the value of that character on that class or permutation. Note that the table may be NULL, in which case the value is computed, or it may be taken from a precalculated charactertable.

FIXME: add table parameter

EXAMPLES:

```
sage: n = 3
sage: m = matrix([[symmetrica.charvalue(irred, cls) for cls in Partitions(n)] for␣
↪irred in Partitions(n)]); m
[ 1  1  1]
[-1  0  2]
[ 1 -1  1]
sage: m == symmetrica.chartafel(n)
True
sage: n = 4
sage: m = matrix([[symmetrica.charvalue(irred, cls) for cls in Partitions(n)] for␣
↪irred in Partitions(n)])
sage: m == symmetrica.chartafel(n)
True
```

sage.libs.symmetrica.symmetrica.**compute_elmsym_with_alphabet_symmetrica**(*n*, *length*, *alphabet='x'*)

computes the expansion of a elementary symmetric function labeled by a INTEGER number as a POLYNOM erg. The object number may also be a PARTITION or a ELM_SYM object. The INTEGER length specifies the length of the alphabet. Both routines are the same.

EXAMPLES: sage: a = symmetrica.compute_elmsym_with_alphabet(2,2); a x0*x1 sage: a.parent() Multivariate Polynomial Ring in x0, x1 over Integer Ring sage: a = symmetrica.compute_elmsym_with_alphabet([2],2); a x0*x1 sage: symmetrica.compute_elmsym_with_alphabet(3,2) 0 sage: symmetrica.compute_elmsym_with_alphabet([3,2,1],2) 0

sage.libs.symmetrica.symmetrica.**compute_homsym_with_alphabet_symmetrica**(*n*, *length*, *alphabet='x'*)

computes the expansion of a homogenous(=complete) symmetric function labeled by a INTEGER number as a POLYNOM erg. The object number may also be a PARTITION or a HOM_SYM object. The INTEGER laenge specifies the length of the alphabet. Both routines are the same.

EXAMPLES: sage: symmetrica.compute_homsym_with_alphabet(3,1,'x') x^3 sage: symmetrica.compute_homsym_with_alphabet([2,1],1,'x') x^3 sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'x') x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'a,b') a^3 + 2*a^2*b + 2*a*b^2 + b^3 sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'x').parent() Multivariate Polynomial Ring in x0, x1 over Integer Ring

sage.libs.symmetrica.symmetrica.**compute_monomial_with_alphabet_symmetrica**(*n*, *length*, *alphabet='x'*)

computes the expansion of a monomial symmetric function labeled by a PARTITION number as a POLYNOM erg. The INTEGER laenge specifies the length of the alphabet.

EXAMPLES: sage: symmetrica.compute_monomial_with_alphabet([2,1],2,'x') x0^2*x1 + x0*x1^2 sage: symmetrica.compute_monomial_with_alphabet([1,1,1],2,'x') 0 sage: symmetrica.compute_monomial_with_alphabet(2,2,'x') x0^2 + x1^2 sage: symmetrica.compute_monomial_with_alphabet(2,2,'a,b') a^2 + b^2 sage: symmetrica.compute_monomial_with_alphabet(2,2,'x').parent() Multivariate Polynomial Ring in x0, x1 over Integer Ring

sage.libs.symmetrica.symmetrica.**compute_powsym_with_alphabet_symmetrica**(*n*, *length*, *alphabet='x'*)

computes the expansion of a power symmetric function labeled by a INTEGER label or by a PARTITION label or a POW_SYM label as a POLYNOM erg. The INTEGER laenge specifies the length of the alphabet.

EXAMPLES: sage: symmetrica.compute_powsym_with_alphabet(2,2,'x') x0^2 + x1^2 sage: symmetrica.compute_powsym_with_alphabet(2,2,'x').parent() Multivariate Polynomial Ring in x0, x1 over Integer Ring sage: symmetrica.compute_powsym_with_alphabet([2],2,'x') x0^2 + x1^2 sage: symmetrica.compute_powsym_with_alphabet([2],2,'a,b') a^2 + b^2 sage: symmetrica.compute_powsym_with_alphabet([2,1],2,'a,b') a^3 + a^2*b + a*b^2 + b^3

sage.libs.symmetrica.symmetrica.**compute_schur_with_alphabet_det_symmetrica**(*part*, *length*, *alphabet='x'*)

EXAMPLES: sage: symmetrica.compute_schur_with_alphabet_det(2,2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet_det([2],2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2,'y') y0^2 + y0*y1 + y1^2 sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2,'a,b') a^2 + a*b + b^2

sage.libs.symmetrica.symmetrica.**compute_schur_with_alphabet_symmetrica**(*part*, *length*, *alphabet='x'*)

Computes the expansion of a schurfunction labeled by a partition PART as a POLYNOM erg. The INTEGER length specifies the length of the alphabet.

EXAMPLES: sage: symmetrica.compute_schur_with_alphabet(2,2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet([2],2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2) x0^2 + x0*x1 + x1^2 sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2,'y') y0^2 + y0*y1 + y1^2 sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2,'a,b') a^2 + a*b + b^2 sage: symmetrica.compute_schur_with_alphabet([2,1],1,'x') 0

sage.libs.symmetrica.symmetrica.**dimension_schur_symmetrica**(*s*)

you enter a SCHUR object a, and the result is the dimension of the corresponding representation of the symmetric group sn.

sage.libs.symmetrica.symmetrica.**dimension_symmetrization_symmetrica**(*n*, *part*)

computes the dimension of the degree of a irreducible representation of the GL_n, n is a INTEGER object, labeled by the PARTITION object a.

sage.libs.symmetrica.symmetrica.**divdiff_perm_schubert_symmetrica**(*perm*, *a*)

Returns the result of applying the divided difference operator δ_i to a where a is either a permutation or a Schubert polynomial over QQ.

EXAMPLES: sage: symmetrica.divdiff_perm_schubert([2,3,1], [3,2,1]) X[2, 1] sage: symmetrica.divdiff_perm_schubert([3,1,2], [3,2,1]) X[1, 3, 2] sage: symmetrica.divdiff_perm_schubert([3,2,4,1], [3,2,1]) Traceback (most recent call last): ... ValueError: cannot apply delta_{[3, 2, 4, 1]} to a (= [3, 2, 1])

sage.libs.symmetrica.symmetrica.**divdiff_schubert_symmetrica**(*i*, *a*)

Returns the result of applying the divided difference operator δ_i to a where a is either a permutation or a Schubert polynomial over QQ.

EXAMPLES: sage: symmetrica.divdiff_schubert(1, [3,2,1]) X[2, 3, 1] sage: symmetrica.divdiff_schubert(2, [3,2,1]) X[3, 1, 2] sage: symmetrica.divdiff_schubert(3, [3,2,1]) Traceback (most recent call last): ... ValueError: cannot apply delta_{3} to a (= [3, 2, 1])

sage.libs.symmetrica.symmetrica.**end**()

sage.libs.symmetrica.symmetrica.**gupta_nm_symmetrica**(*n*, *m*)

this routine computes the number of partitions of n with maximal part m. The result is erg. The input n,m must be INTEGER objects. The result is freed first to an empty object. The result must be a different from m and n.

sage.libs.symmetrica.symmetrica.**gupta_tafel_symmetrica** (*max*)
it computes the table of the above values. The entry n,m is the result of gupta_nm. mat is freed first. max must be an INTEGER object, it is the maximum weight for the partitions. max must be different from result.

sage.libs.symmetrica.symmetrica.**hall_littlewood_symmetrica** (*part*)
computes the so called Hall Littlewood Polynomials, i.e. a SCHUR object, whose coefficient are polynomials in one variable. The method, which is used for the computation is described in the paper: A.O. Morris The Characters of the group GL(n,q) Math Zeitschr 81, 112-123 (1963)

sage.libs.symmetrica.symmetrica.**kostka_number_symmetrica** (*shape*, *content*)
computes the kostkanumber, i.e. the number of tableaux of given shape, which is a PARTITION object, and of given content, which also is a PARTITION object, or a VECTOR object with INTEGER entries. The result is an INTEGER object, which is freed to an empty object at the beginning. The shape could also be a SKEWPARTITION object, then we compute the number of skewtableaux of the given shape.

EXAMPLES:

```
sage: symmetrica.kostka_number([2,1],[1,1,1])
2
sage: symmetrica.kostka_number([1,1,1],[1,1,1])
1
sage: symmetrica.kostka_number([3],[1,1,1])
1
```

sage.libs.symmetrica.symmetrica.**kostka_tab_symmetrica** (*shape*, *content*)
computes the list of tableaux of given shape and content. shape is a PARTITION object or a SKEWPARTITION object and content is a PARTITION object or a VECTOR object with INTEGER entries, the result becomes a LIST object whose entries are the computed TABLEAUX object.

EXAMPLES:

```
sage: symmetrica.kostka_tab([3],[1,1,1])
[[[1, 2, 3]]]
sage: symmetrica.kostka_tab([2,1],[1,1,1])
[[[1, 2], [3]], [[1, 3], [2]]]
sage: symmetrica.kostka_tab([1,1,1],[1,1,1])
[[[1], [2], [3]]]
sage: symmetrica.kostka_tab([[2,2,1],[1,1]],[1,1,1])
[[[None, 1], [None, 2], [3]],
 [[None, 1], [None, 3], [2]],
 [[None, 2], [None, 3], [1]]]
sage: symmetrica.kostka_tab([[2,2],[1]],[1,1,1])
[[[None, 1], [2, 3]], [[None, 2], [1, 3]]]
```

sage.libs.symmetrica.symmetrica.**kostka_tafel_symmetrica** (*n*)
Returns the table of Kostka numbers of weight n.

EXAMPLES:

```
sage: symmetrica.kostka_tafel(1)
[1]

sage: symmetrica.kostka_tafel(2)
[1 0]
[1 1]

sage: symmetrica.kostka_tafel(3)
[1 0 0]
[1 1 0]
```

```
[1 2 1]

sage: symmetrica.kostka_tafel(4)
[1 0 0 0 0]
[1 1 0 0 0]
[1 1 1 0 0]
[1 2 1 1 0]
[1 3 2 3 1]

sage: symmetrica.kostka_tafel(5)
[1 0 0 0 0 0 0]
[1 1 0 0 0 0 0]
[1 1 1 0 0 0 0]
[1 2 1 1 0 0 0]
[1 2 2 1 1 0 0]
[1 3 3 3 2 1 0]
[1 4 5 6 5 4 1]
```

sage.libs.symmetrica.symmetrica.**kranztafel_symmetrica**(*a*, *b*)

you enter the INTEGER objects, say a and b, and res becomes a MATRIX object, the charactertable of S_b wr S_a, co becomes a VECTOR object of classorders and cl becomes a VECTOR object of the classlabels.

EXAMPLES:

```
sage: (a,b,c) = symmetrica.kranztafel(2,2)
sage: a
[ 1 -1  1 -1  1]
[ 1  1  1  1  1]
[-1  1  1 -1  1]
[ 0  0  2  0 -2]
[-1 -1  1  1  1]
sage: b
[2, 2, 1, 2, 1]
sage: for m in c: print(m)
...
[0 0]
[0 1]
[0 0]
[1 0]
[0 2]
[0 0]
[1 1]
[0 0]
[2 0]
[0 0]
```

sage.libs.symmetrica.symmetrica.**mult_monomial_monomial_symmetrica**(*m1*, *m2*)

sage.libs.symmetrica.symmetrica.**mult_schubert_schubert_symmetrica**(*a*, *b*)

Multiplies the Schubert polynomials a and b.

EXAMPLES: sage: symmetrica.mult_schubert_schubert([3,2,1], [3,2,1]) X[5, 3, 1, 2, 4]

sage.libs.symmetrica.symmetrica.**mult_schubert_variable_symmetrica**(*a*, *i*)

Returns the product of a and x_i. Note that indexing with i starts at 1.

EXAMPLES: sage: symmetrica.mult_schubert_variable([3,2,1], 2) X[3, 2, 4, 1] sage: symmetrica.mult_schubert_variable([3,2,1], 4) X[3, 2, 1, 4, 6, 5] - X[3, 2, 1, 5, 4]

sage.libs.symmetrica.symmetrica.**mult_schur_schur_symmetrica**(*s1*, *s2*) `sage.libs.symmetrica.symmetrica.`**`ndg_symmetrica`**`(`*part*, *perm*`)`

`sage.libs.symmetrica.symmetrica.`**`newtrans_symmetrica`**`(`*perm*`)`
computes the decomposition of a schubertpolynomial labeled by the permutation perm, as a sum of Schurfunction. FIXME!

`sage.libs.symmetrica.symmetrica.`**`odd_to_strict_part_symmetrica`**`(`*part*`)`
implements the bijection between partitions with odd parts and strict partitions. input is a VECTOR type partition, the result is a partition of the same weight with different parts.

`sage.libs.symmetrica.symmetrica.`**`odg_symmetrica`**`(`*part*, *perm*`)`
Calculates the irreduzible matrix representation D^part(perm), which consists of real numbers.

REFERENCE: G. James/ A. Kerber: Representation Theory of the Symmetric Group. Addison/Wesley 1981. pp. 127-129.

`sage.libs.symmetrica.symmetrica.`**`outerproduct_schur_symmetrica`**`(`*parta*, *partb*`)`
you enter two PARTITION objects, and the result is a SCHUR object, which is the expansion of the product of the two schurfunctions, labbeled by the two PARTITION objects parta and partb. Of course this can also be interpreted as the decomposition of the outer tensor product of two irreducibe representations of the symmetric group.

EXAMPLES: sage: symmetrica.outerproduct_schur([2],[2]) s[2, 2] + s[3, 1] + s[4]

`sage.libs.symmetrica.symmetrica.`**`part_part_skewschur_symmetrica`**`(`*outer*, *inner*`)`
Returns the skew schur function s_{outer/inner}

EXAMPLES: sage: symmetrica.part_part_skewschur([3,2,1],[2,1]) s[1, 1, 1] + 2*s[2, 1] + s[3]

`sage.libs.symmetrica.symmetrica.`**`plethysm_symmetrica`**`(`*outer*, *inner*`)`

`sage.libs.symmetrica.symmetrica.`**`q_core_symmetrica`**`(`*part*, *d*`)`
computes the q-core of a PARTITION object part. This is the remaining partition (=res) after removing of all hooks of length d (= INTEGER object). The result may be an empty object, if the whole partition disappears.

`sage.libs.symmetrica.symmetrica.`**`random_partition_symmetrica`**`(`*n*`)`
returns a random partition p of the entered weight w. w must be an INTEGER object, p becomes a PARTITION object. Type of partition is VECTOR . Its the algorithm of Nijnhuis Wilf p.76

`sage.libs.symmetrica.symmetrica.`**`scalarproduct_schubert_symmetrica`**`(`*a*, *b*`)`

EXAMPLES: sage: symmetrica.scalarproduct_schubert([3,2,1], [3,2,1]) X[1, 3, 5, 2, 4] sage: symmetrica.scalarproduct_schubert([3,2,1], [2,1,3]) X[1, 2, 4, 3]

`sage.libs.symmetrica.symmetrica.`**`scalarproduct_schur_symmetrica`**`(`*s1*, *s2*`)`

`sage.libs.symmetrica.symmetrica.`**`schur_schur_plet_symmetrica`**`(`*outer*, *inner*`)`

`sage.libs.symmetrica.symmetrica.`**`sdg_symmetrica`**`(`*part*, *perm*`)`
Calculates the irreduzible matrix representation D^part(perm), which consists of rational numbers.

REFERENCE: G. James/ A. Kerber: Representation Theory of the Symmetric Group. Addison/Wesley 1981. pp. 124-126.

`sage.libs.symmetrica.symmetrica.`**`specht_dg_symmetrica`**`(`*part*, *perm*`)`

`sage.libs.symmetrica.symmetrica.`**`start`**`()`

`sage.libs.symmetrica.symmetrica.`**`strict_to_odd_part_symmetrica`**`(`*part*`)`
implements the bijection between strict partitions and partitions with odd parts. input is a VECTOR type partition, the result is a partition of the same weight with only odd parts.

`sage.libs.symmetrica.symmetrica.`**`t_ELMSYM_HOMSYM_symmetrica`**`(`*elmsym*`)`

`sage.libs.symmetrica.symmetrica.`**`t_ELMSYM_MONOMIAL_symmetrica`**`(`*elmsym*`)` sage.libs.symmetrica.symmetrica.**t_ELMSYM_POWSYM_symmetrica** (*elmsym*)

sage.libs.symmetrica.symmetrica.**t_ELMSYM_SCHUR_symmetrica** (*elmsym*)

sage.libs.symmetrica.symmetrica.**t_HOMSYM_ELMSYM_symmetrica** (*homsym*)

sage.libs.symmetrica.symmetrica.**t_HOMSYM_MONOMIAL_symmetrica** (*homsym*)

sage.libs.symmetrica.symmetrica.**t_HOMSYM_POWSYM_symmetrica** (*homsym*)

sage.libs.symmetrica.symmetrica.**t_HOMSYM_SCHUR_symmetrica** (*homsym*)

sage.libs.symmetrica.symmetrica.**t_MONOMIAL_ELMSYM_symmetrica** (*monomial*)

sage.libs.symmetrica.symmetrica.**t_MONOMIAL_HOMSYM_symmetrica** (*monomial*)

sage.libs.symmetrica.symmetrica.**t_MONOMIAL_POWSYM_symmetrica** (*monomial*)

sage.libs.symmetrica.symmetrica.**t_MONOMIAL_SCHUR_symmetrica** (*monomial*)

sage.libs.symmetrica.symmetrica.**t_POLYNOM_ELMSYM_symmetrica** (*p*)
Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the elementary basis.

sage.libs.symmetrica.symmetrica.**t_POLYNOM_MONOMIAL_symmetrica** (*p*)
Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the monomial basis.

sage.libs.symmetrica.symmetrica.**t_POLYNOM_POWER_symmetrica** (*p*)
Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the power sum basis.

sage.libs.symmetrica.symmetrica.**t_POLYNOM_SCHUBERT_symmetrica** (*a*)
Converts a multivariate polynomial a to a Schubert polynomial.

EXAMPLES: sage: R.<x1,x2,x3> = QQ[] sage: w0 = x1^2*x2 sage: symmetrica.t_POLYNOM_SCHUBERT(w0) X[3, 2, 1]

sage.libs.symmetrica.symmetrica.**t_POLYNOM_SCHUR_symmetrica** (*p*)
Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the Schur basis.

sage.libs.symmetrica.symmetrica.**t_POWSYM_ELMSYM_symmetrica** (*powsym*)

sage.libs.symmetrica.symmetrica.**t_POWSYM_HOMSYM_symmetrica** (*powsym*)

sage.libs.symmetrica.symmetrica.**t_POWSYM_MONOMIAL_symmetrica** (*powsym*)

sage.libs.symmetrica.symmetrica.**t_POWSYM_SCHUR_symmetrica** (*powsym*)

sage.libs.symmetrica.symmetrica.**t_SCHUBERT_POLYNOM_symmetrica** (*a*)
Converts a Schubert polynomial to a ‘regular’ multivariate polynomial.

EXAMPLES: sage: symmetrica.t_SCHUBERT_POLYNOM([3,2,1]) x0^2*x1

sage.libs.symmetrica.symmetrica.**t_SCHUR_ELMSYM_symmetrica** (*schur*)

sage.libs.symmetrica.symmetrica.**t_SCHUR_HOMSYM_symmetrica** (*schur*)

sage.libs.symmetrica.symmetrica.**t_SCHUR_MONOMIAL_symmetrica** (*schur*)

sage.libs.symmetrica.symmetrica.**t_SCHUR_POWSYM_symmetrica** (*schur*)

sage.libs.symmetrica.symmetrica.**test_integer** (*x*)
Tests functionality for converting between Sage’s integers and symmetrica’s integers.

EXAMPLES:

```
sage: from sage.libs.symmetrica.symmetrica import test_integer
sage: test_integer(1)
1
sage: test_integer(-1)
```

```
-1
sage: test_integer(2^33)
8589934592
sage: test_integer(-2^33)
-8589934592
sage: test_integer(2^100)
1267650600228229401496703205376
sage: test_integer(-2^100)
-1267650600228229401496703205376
sage: for i in range(100):
....:     if test_integer(2^i) != 2^i:
....:         print("Failure at {}".format(i))
```

CHAPTER

TWENTYTHREE

UTILITIES FOR SAGE-MPMATH INTERACTION

Also patches some mpmath functions for speed

`sage.libs.mpmath.utils.` **`bitcount`** (*n*)

Bitcount of a Sage Integer or Python int/long.

EXAMPLES:

```
sage: from mpmath.libmp import bitcount
sage: bitcount(0)
0
sage: bitcount(1)
1
sage: bitcount(100)
7
sage: bitcount(-100)
7
sage: bitcount(2r)
2
sage: bitcount(2L)
2
```

`sage.libs.mpmath.utils.` **`call`** (*func*, **args*, ***kwargs*)

Call an mpmath function with Sage objects as inputs and convert the result back to a Sage real or complex number.

By default, a RealNumber or ComplexNumber with the current working precision of mpmath (mpmath.mp.prec) will be returned.

If prec=n is passed among the keyword arguments, the temporary working precision will be set to n and the result will also have this precision.

If parent=P is passed, P.prec() will be used as working precision and the result will be coerced to P (or the corresponding complex field if necessary).

Arguments should be Sage objects that can be coerced into RealField or ComplexField elements. Arguments may also be tuples, lists or dicts (which are converted recursively), or any type that mpmath understands natively (e.g. Python floats, strings for options).

EXAMPLES:

```
sage: import sage.libs.mpmath.all as a
sage: a.mp.prec = 53
sage: a.call(a.erf, 3+4*I)
-120.186991395079 - 27.7503372936239*I
sage: a.call(a.polylog, 2, 1/3+4/5*I)
0.153548951541433 + 0.875114412499637*I
```

```
sage: a.call(a.barnesg, 3+4*I)
-0.000676375932234244 - 0.0000442236140124728*I
sage: a.call(a.barnesg, -4)
0.000000000000000
sage: a.call(a.hyper, [2,3], [4,5], 1/3)
1.10703578162508
sage: a.call(a.hyper, [2,3], [4,(2,3)], 1/3)
1.95762943509305
sage: a.call(a.quad, a.erf, [0,1])
0.486064958112256
sage: a.call(a.gammainc, 3+4*I, 2/3, 1-pi*I, prec=100)
-274.18871130777160922270612331 + 101.59521032382593402947725236*I
sage: x = (3+4*I).n(100)
sage: y = (2/3).n(100)
sage: z = (1-pi*I).n(100)
sage: a.call(a.gammainc, x, y, z, prec=100)
-274.18871130777160922270612331 + 101.59521032382593402947725236*I
sage: a.call(a.erf, infinity)
1.00000000000000
sage: a.call(a.erf, -infinity)
-1.00000000000000
sage: a.call(a.gamma, infinity)
+infinity
sage: a.call(a.polylog, 2, 1/2, parent=RR)
0.582240526465012
sage: a.call(a.polylog, 2, 2, parent=RR)
2.46740110027234 - 2.17758609030360*I
sage: a.call(a.polylog, 2, 1/2, parent=RealField(100))
0.58224052646501250590265632016
sage: a.call(a.polylog, 2, 2, parent=RealField(100))
2.4674011002723396547086227500 - 2.1775860903036021305006888982*I
sage: a.call(a.polylog, 2, 1/2, parent=CC)
0.582240526465012
sage: type(_)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: a.call(a.polylog, 2, 1/2, parent=RDF)
0.5822405264650125
sage: type(_)
<type 'sage.rings.real_double.RealDoubleElement'>
```

Check that trac ticket #11885 is fixed:

```
sage: a.call(a.ei, 1.0r, parent=float)
1.8951178163559366
```

Check that trac ticket #14984 is fixed:

```
sage: a.call(a.log, -1.0r, parent=float)
3.141592653589793j
```

`sage.libs.mpmath.utils.`**`from_man_exp`** (*man*, *exp*, *prec=0*, *rnd='d'*)

Create normalized mpf value tuple from mantissa and exponent.

With prec > 0, rounds the result in the desired direction if necessary.

EXAMPLES:

```
sage: from mpmath.libmp import from_man_exp
sage: from_man_exp(-6, -1)
```

```
(1, 3, 0, 2)
sage: from_man_exp(-6, -1, 1, 'd')
(1, 1, 1, 1)
sage: from_man_exp(-6, -1, 1, 'u')
(1, 1, 2, 1)
```

sage.libs.mpmath.utils.**isqrt**(*n*)

Square root (rounded to floor) of a Sage Integer or Python int/long. The result is a Sage Integer.

EXAMPLES:

```
sage: from mpmath.libmp import isqrt
sage: isqrt(0)
0
sage: isqrt(100)
10
sage: isqrt(10)
3
sage: isqrt(10r)
3
sage: isqrt(10L)
3
```

sage.libs.mpmath.utils.**mpmath_to_sage**(*x*, *prec*)

Convert any mpmath number (mpf or mpc) to a Sage RealNumber or ComplexNumber of the given precision.

EXAMPLES:

```
sage: import sage.libs.mpmath.all as a
sage: a.mpmath_to_sage(a.mpf('2.5'), 53)
2.50000000000000
sage: a.mpmath_to_sage(a.mpc('2.5','-3.5'), 53)
2.50000000000000 - 3.50000000000000*I
sage: a.mpmath_to_sage(a.mpf('inf'), 53)
+infinity
sage: a.mpmath_to_sage(a.mpf('-inf'), 53)
-infinity
sage: a.mpmath_to_sage(a.mpf('nan'), 53)
NaN
sage: a.mpmath_to_sage(a.mpf('0'), 53)
0.000000000000000
```

A real example:

```
sage: RealField(100)(pi)
3.1415926535897932384626433833
sage: t = RealField(100)(pi)._mpmath_(); t
mpf('3.1415926535897932')
sage: a.mpmath_to_sage(t, 100)
3.1415926535897932384626433833
```

We can ask for more precision, but the result is undefined:

```
sage: a.mpmath_to_sage(t, 140) # random
3.1415926535897932384626433832793333156440
sage: ComplexField(140)(pi)
3.1415926535897932384626433832795028841972
```

A complex example:

```
sage: ComplexField(100)([0, pi])
3.1415926535897932384626433833*I
sage: t = ComplexField(100)([0, pi])._mpmath_(); t
mpc(real='0.0', imag='3.1415926535897932')
sage: sage.libs.mpmath.all.mpmath_to_sage(t, 100)
3.1415926535897932384626433833*I
```

Again, we can ask for more precision, but the result is undefined:

```
sage: sage.libs.mpmath.all.mpmath_to_sage(t, 140) # random
3.1415926535897932384626433832793333156440*I
sage: ComplexField(140)([0, pi])
3.1415926535897932384626433832795028841972*I
```

sage.libs.mpmath.utils.**normalize**(*sign*, *man*, *exp*, *bc*, *prec*, *rnd*)

Create normalized mpf value tuple from full list of components.

EXAMPLES:

```
sage: from mpmath.libmp import normalize
sage: normalize(0, 4, 5, 3, 53, 'n')
(0, 1, 7, 1)
```

sage.libs.mpmath.utils.**sage_to_mpmath**(*x*, *prec*)

Convert any Sage number that can be coerced into a RealNumber or ComplexNumber of the given precision into an mpmath mpf or mpc. Integers are currently converted to int.

Lists, tuples and dicts passed as input are converted recursively.

EXAMPLES:

```
sage: import sage.libs.mpmath.all as a
sage: a.mp.dps = 15
sage: print(a.sage_to_mpmath(2/3, 53))
0.666666666666667
sage: print(a.sage_to_mpmath(2./3, 53))
0.666666666666667
sage: print(a.sage_to_mpmath(3+4*I, 53))
(3.0 + 4.0j)
sage: print(a.sage_to_mpmath(1+pi, 53))
4.14159265358979
sage: a.sage_to_mpmath(infinity, 53)
mpf('+inf')
sage: a.sage_to_mpmath(-infinity, 53)
mpf('-inf')
sage: a.sage_to_mpmath(NaN, 53)
mpf('nan')
sage: a.sage_to_mpmath(0, 53)
0
sage: a.sage_to_mpmath([0.5, 1.5], 53)
[mpf('0.5'), mpf('1.5')]
sage: a.sage_to_mpmath((0.5, 1.5), 53)
(mpf('0.5'), mpf('1.5'))
sage: a.sage_to_mpmath({'n':0.5}, 53)
{'n': mpf('0.5')}
```

CHAPTER

TWENTYFOUR

VICTOR SHOUP'S NTL C++ LIBRARY

Sage provides an interface to Victor Shoup's C++ library NTL. Features of this library include *incredibly fast* arithmetic with polynomials and asymptotically fast factorization of polynomials.

CHAPTER

TWENTYFIVE

THE ELLIPTIC CURVE METHOD FOR INTEGER FACTORIZATION (ECM)

Sage includes GMP-ECM, which is a highly optimized implementation of Lenstra's elliptic curve factorization method. See http://ecm.gforge.inria.fr/ for more about GMP-ECM. This file provides a Cython interface to the GMP-ECM library.

AUTHORS:

- Robert L Miller (2008-01-21): library interface (clone of ecmfactor.c)
- Jeroen Demeyer (2012-03-29): signal handling, documentation
- Paul Zimmermann (2011-05-22) – added input/output of sigma

EXAMPLES:

```
sage: from sage.libs.libecm import ecmfactor
sage: result = ecmfactor(999, 0.00)
sage: result[0] and (result[1] in [27, 37, 999])
True
sage: result = ecmfactor(999, 0.00, verbose=True)
Performing one curve with B1=0
Found factor in step 1: ...
sage: result[0] and (result[1] in [27, 37, 999])
True
sage: ecmfactor(2^128+1,1000,sigma=227140902)
(True, 5704689200685129054721, 227140902)
```

sage.libs.libecm.**ecmfactor**(*number*, *B1*, *verbose=False*, *sigma=0*)

Try to find a factor of a positive integer using ECM (Elliptic Curve Method). This function tries one elliptic curve.

INPUT:

- `number` – positive integer to be factored
- `B1` – bound for step 1 of ECM
- `verbose` (default: False) – print some debugging information

OUTPUT:

Either `(False,None)` if no factor was found, or `(True,f)` if the factor `f` was found.

EXAMPLES:

```
sage: from sage.libs.libecm import ecmfactor
```

This number has a small factor which is easy to find for ECM:

```
sage: N = 2^167 - 1
sage: factor(N)
2349023 * 79638304766856507377778616296087448490695649
sage: ecmfactor(N, 2e5)
(True, 2349023, ...)
```

If a factor was found, we can reproduce the factorization with the same sigma value:

```
sage: N = 2^167 - 1
sage: ecmfactor(N, 2e5, sigma=1473308225)
(True, 2349023, 1473308225)
```

With a smaller B1 bound, we may or may not succeed:

```
sage: ecmfactor(N, 1e2)  # random
(False, None)
```

The following number is a Mersenne prime, so we don't expect to find any factors (there is an extremely small chance that we get the input number back as factorization):

```
sage: N = 2^127 - 1
sage: N.is_prime()
True
sage: ecmfactor(N, 1e3)
(False, None)
```

If we have several small prime factors, it is possible to find a product of primes as factor:

```
sage: N = 2^179 - 1
sage: factor(N)
359 * 1433 * 1489459109360039866456940197095433721664951999121
sage: ecmfactor(N, 1e3)  # random
(True, 514447, 3475102204)
```

We can ask for verbose output:

```
sage: N = 12^97 - 1
sage: factor(N)
11 *␣
↪43570062353753446053455610056679740005056966111842089407838902783209959981593077811330507328832
sage: ecmfactor(N, 100, verbose=True)
Performing one curve with B1=100
Found factor in step 1: 11
(True, 11, ...)
sage: ecmfactor(N/11, 100, verbose=True)
Performing one curve with B1=100
Found no factor.
(False, None)
```

TESTS:

Check that `ecmfactor` can be interrupted (factoring a large prime number):

```
sage: alarm(0.5); ecmfactor(2^521-1, 1e7)
Traceback (most recent call last):
...
AlarmInterrupt
```

Some special cases:

```
sage: ecmfactor(1, 100)
(True, 1, ...)
sage: ecmfactor(0, 100)
Traceback (most recent call last):
...
ValueError: Input number (0) must be positive
```

CHAPTER

TWENTYSIX

AN INTERFACE TO ANDERS BUCH'S LITTLEWOOD-RICHARDSON CALCULATOR LRCALC

The "Littlewood-Richardson Calculator" is a C library for fast computation of Littlewood-Richardson (LR) coefficients and products of Schubert polynomials. It handles single LR coefficients, products of and coproducts of Schur functions, skew Schur functions, and fusion products. All of the above are achieved by counting LR (skew)-tableaux (also called Yamanouchi (skew)-tableaux) of appropriate shape and content by iterating through them. Additionally, `lrcalc` handles products of Schubert polynomials.

The web page of `lrcalc` is http://math.rutgers.edu/~asbuch/lrcalc/.

The following describes the Sage interface to this library.

EXAMPLES:

```
sage: import sage.libs.lrcalc.lrcalc as lrcalc
```

Compute a single Littlewood-Richardson coefficient:

```
sage: lrcalc.lrcoef([3,2,1],[2,1],[2,1])
2
```

Compute a product of Schur functions; return the coefficients in the Schur expansion:

```
sage: lrcalc.mult([2,1], [2,1])
{[2, 2, 1, 1]: 1,
 [2, 2, 2]: 1,
 [3, 1, 1, 1]: 1,
 [3, 2, 1]: 2,
 [3, 3]: 1,
 [4, 1, 1]: 1,
 [4, 2]: 1}
```

Same product, but include only partitions with at most 3 rows. This corresponds to computing in the representation ring of gl(3):

```
sage: lrcalc.mult([2,1], [2,1], 3)
{[2, 2, 2]: 1, [3, 2, 1]: 2, [3, 3]: 1, [4, 1, 1]: 1, [4, 2]: 1}
```

We can also compute the fusion product, here for sl(3) and level 2:

```
sage: lrcalc.mult([3,2,1], [3,2,1], 3,2)
{[4, 4, 4]: 1, [5, 4, 3]: 1}
```

Compute the expansion of a skew Schur function:

```
sage: lrcalc.skew([3,2,1],[2,1])
{[1, 1, 1]: 1, [2, 1]: 2, [3]: 1}
```

Compute the coproduct of a Schur function:

```
sage: lrcalc.coprod([3,2,1])
{([1, 1, 1], [2, 1]): 1,
 ([2, 1], [2, 1]): 2,
 ([2, 1], [3]): 1,
 ([2, 1, 1], [1, 1]): 1,
 ([2, 1, 1], [2]): 1,
 ([2, 2], [1, 1]): 1,
 ([2, 2], [2]): 1,
 ([2, 2, 1], [1]): 1,
 ([3, 1], [1, 1]): 1,
 ([3, 1], [2]): 1,
 ([3, 1, 1], [1]): 1,
 ([3, 2], [1]): 1,
 ([3, 2, 1], []): 1}
```

Multiply two Schubert polynomials:

```
sage: lrcalc.mult_schubert([4,2,1,3], [1,4,2,5,3])
{[4, 5, 1, 3, 2]: 1,
 [5, 3, 1, 4, 2]: 1,
 [5, 4, 1, 2, 3]: 1,
 [6, 2, 1, 4, 3, 5]: 1}
```

Same product, but include only permutations of 5 elements in the result. This corresponds to computing in the cohomology ring of Fl(5):

```
sage: lrcalc.mult_schubert([4,2,1,3], [1,4,2,5,3], 5)
{[4, 5, 1, 3, 2]: 1, [5, 3, 1, 4, 2]: 1, [5, 4, 1, 2, 3]: 1}
```

List all Littlewood-Richardson tableaux of skew shape μ/ν; in this example $\mu = [3,2,1]$ and $\nu = [2,1]$. Specifying a third entry $maxrows$ restricts the alphabet to $\{1, 2, \ldots, maxrows\}$:

```
sage: list(lrcalc.lrskew([3,2,1],[2,1]))
[[[None, None, 1], [None, 1], [1]], [[None, None, 1], [None, 1], [2]],
[[None, None, 1], [None, 2], [1]], [[None, None, 1], [None, 2], [3]]]

sage: list(lrcalc.lrskew([3,2,1],[2,1],maxrows=2))
[[[None, None, 1], [None, 1], [1]], [[None, None, 1], [None, 1], [2]], [[None, None,␣
↪1], [None, 2], [1]]]
```

Todo

use this library in the `SymmetricFunctions` code, to make it easy to apply it to linear combinations of Schur functions.

See also:

- *lrcoef()*
- *mult()*
- *coprod()*

- `skew()`
- `lrskew()`
- `mult_schubert()`

Underlying algorithmic in lrcalc

Here is some additional information regarding the main low-level C-functions in *lrcalc*. Given two partitions `outer` and `inner` with `inner` contained in `outer`, the function:

```
skewtab *st_new(vector *outer, vector *inner, vector *conts, int maxrows)
```

constructs and returns the (lexicographically) first LR skew tableau of shape `outer / inner`. Further restrictions can be imposed using `conts` and `maxrows`.

Namely, the integer `maxrows` is a bound on the integers that can be put in the tableau. The name is chosen because this will limit the partitions in the output of `skew()` or `mult()` to partitions with at most this number of rows.

The vector `conts` is the content of an empty tableau(!!). More precisely, this vector is added to the usual content of a tableau whenever the content is needed. This affects which tableaux are considered LR tableaux (see `mult()` below). `conts` may also be the `NULL` pointer, in which case nothing is added.

The other function:

```
int *st_next(skewtab *st)
```

computes in place the (lexicographically) next skew tableau with the same constraints, or returns 0 if `st` is the last one.

For a first example, see the `skew()` function code in the `lrcalc` source code. We want to compute a skew schur function, so create a skew LR tableau of the appropriate shape with `st_new` (with `conts = NULL`), then iterate through all the LR tableaux with `st_next()`. For each skew tableau, we use that `st->conts` is the content of the skew tableau, find this shape in the `res` hash table and add one to the value.

For a second example, see `mult(vector *sh1,vector *sh2,maxrows)`. Here we call `st_new()` with the shape `sh1 / (0)` and use `sh2` as the `conts` argument. The effect of using `sh2` in this way is that `st_next` will iterate through semistandard tableaux T of shape `sh1` such that the following tableau:

```
     111111
     22222    <--- minimal tableau of shape sh2
     333
*****
**T**
****
**
```

is a LR skew tableau, and `st->conts` contains the content of the combined tableaux.

More generally, `st_new(outer,inner,conts,maxrows)` and `st_next` can be used to compute the Schur expansion of the product `S_{outer/inner} * S_conts`, restricted to partitions with at most `maxrows` rows.

AUTHORS:

- Mike Hansen (2010): core of the interface
- Anne Schilling, Nicolas M. Thiéry, and Anders Buch (2011): fusion product, iterating through LR tableaux, finalization, documentation

sage.libs.lrcalc.lrcalc.**coprod**(*part*, *all=0*)

Compute the coproduct of a Schur function.

Return a linear combination of pairs of partitions representing the coproduct of the Schur function given by the partition `part`.

INPUT:

- `part` – a partition.
- `all` – an integer.

If `all` is non-zero then all terms are included in the result. If `all` is zero, then only pairs of partitions `(part1,part2)` for which the weight of `part1` is greater than or equal to the weight of `part2` are included; the rest of the coefficients are redundant because Littlewood-Richardson coefficients are symmetric.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import coprod
sage: sorted(coprod([2,1]).items())
[(([1, 1], [1]), 1), (([2], [1]), 1), (([2, 1], []), 1)]
```

sage.libs.lrcalc.lrcalc.**lrcoef**(*outer*, *inner1*, *inner2*)

Compute a single Littlewood-Richardson coefficient.

Return the coefficient of `outer` in the product of the Schur functions indexed by `inner1` and `inner2`.

INPUT:

- `outer` – a partition (weakly decreasing list of non-negative integers).
- `inner1` – a partition.
- `inner2` – a partition.

Note: This function converts its inputs into `Partition()` 's. If you don't need these checks and your inputs are valid, then you can use *`lrcoef_unsafe()`*.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import lrcoef
sage: lrcoef([3,2,1], [2,1], [2,1])
2
sage: lrcoef([3,3], [2,1], [2,1])
1
sage: lrcoef([2,1,1,1,1], [2,1], [2,1])
0
```

sage.libs.lrcalc.lrcalc.**lrcoef_unsafe**(*outer*, *inner1*, *inner2*)

Compute a single Littlewood-Richardson coefficient.

Return the coefficient of `outer` in the product of the Schur functions indexed by `inner1` and `inner2`.

INPUT:

- `outer` – a partition (weakly decreasing list of non-negative integers).
- `inner1` – a partition.
- `inner2` – a partition.

Warning: This function does not do any check on its input. If you want to use a safer version, use *lrcoef()*.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import lrcoef_unsafe
sage: lrcoef_unsafe([3,2,1], [2,1], [2,1])
2
sage: lrcoef_unsafe([3,3], [2,1], [2,1])
1
sage: lrcoef_unsafe([2,1,1,1,1], [2,1], [2,1])
0
```

sage.libs.lrcalc.lrcalc.**lrskew**(*outer*, *inner*, *weight=None*, *maxrows=0*)

Return the skew LR tableaux of shape `outer / inner`.

INPUT:

- `outer` – a partition.
- `inner` – a partition.
- `weight` – a partition (optional).
- `maxrows` – an integer (optional).

OUTPUT: a list of `SkewTableau`x. This will change to an iterator over such skew tableaux once Cython will support the ``yield`` statement. Specifying a third entry $maxrows$ restricts the alphabet to $\{1, 2, \ldots, maxrows\}$. Specifying $weight$ returns only those tableaux of given content/weight.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import lrskew
sage: for st in lrskew([3,2,1],[2]):
...       st.pp()
.  .  1
1  1
2
.  .  1
1  2
2
.  .  1
1  2
3

sage: for st in lrskew([3,2,1],[2], maxrows=2):
...       st.pp()
.  .  1
1  1
2
.  .  1
1  2
2

sage: lrskew([3,2,1],[2], weight=[3,1])
[[[None, None, 1], [1, 1], [2]]]
```

sage.libs.lrcalc.lrcalc.**mult** (*part1*, *part2*, *maxrows=None*, *level=None*, *quantum=None*)

Compute a product of two Schur functions.

Return the product of the Schur functions indexed by the partitions `part1` and `part2`.

INPUT:

- `part1` – a partition
- `part2` – a partition
- `maxrows` – (optional) an integer
- `level` – (optional) an integer
- `quantum` – (optional) an element of a ring

If `maxrows` is specified, then only partitions with at most this number of rows are included in the result.

If both `maxrows` and `level` are specified, then the function calculates the fusion product for $\mathfrak{sl}(\mathrm{maxrows})$ of the given level.

If `quantum` is set, then this returns the product in the quantum cohomology ring of the Grassmannian. In particular, both `maxrows` and `level` need to be specified.

EXAMPLES:

```
    sage: from sage.libs.lrcalc.lrcalc import mult
    sage: mult([2],[])
    {[2]: 1}
    sage: sorted(mult([2],[2]).items())
    [([2, 2], 1), ([3, 1], 1), ([4], 1)]
    sage: sorted(mult([2,1],[2,1]).items())
    [([2, 2, 1, 1], 1), ([2, 2, 2], 1), ([3, 1, 1, 1], 1), ([3, 2, 1], 2), ([3,
→3], 1), ([4, 1, 1], 1), ([4, 2], 1)]
    sage: sorted(mult([2,1],[2,1],maxrows=2).items())
    [([3, 3], 1), ([4, 2], 1)]
    sage: mult([2,1],[3,2,1],3)
    {[3, 3, 3]: 1, [4, 3, 2]: 2, [4, 4, 1]: 1, [5, 2, 2]: 1, [5, 3, 1]: 1}
    sage: mult([2,1],[2,1],3,3)
    {[2, 2, 2]: 1, [3, 2, 1]: 2, [3, 3]: 1, [4, 1, 1]: 1}
    sage: mult([2,1],[2,1],None,3)
    Traceback (most recent call last):
    ...
    ValueError: maxrows needs to be specified if you specify the level

The quantum product::

    sage: q = polygen(QQ, 'q')
    sage: sorted(mult([1],[2,1], 2, 2, quantum=q).items())
    [([], q), ([2, 2], 1)]
    sage: sorted(mult([2,1],[2,1], 2, 2, quantum=q).items())
    [([1, 1], q), ([2], q)]

    sage: mult([2,1],[2,1], quantum=q)
    Traceback (most recent call last):
    ...
    ValueError: missing parameters maxrows or level
```

sage.libs.lrcalc.lrcalc.**mult_schubert** (*w1*, *w2*, *rank=0*)

Compute a product of two Schubert polynomials.

Return a linear combination of permutations representing the product of the Schubert polynomials indexed by the permutations w1 and w2 .

INPUT:

- w1 – a permutation.
- w2 – a permutation.
- rank – an integer.

If rank is non-zero, then only permutations from the symmetric group $S(\mathrm{rank})$ are included in the result.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import mult_schubert
sage: result = mult_schubert([3, 1, 5, 2, 4], [3, 5, 2, 1, 4])
sage: sorted(result.items())
[([5, 4, 6, 1, 2, 3], 1), ([5, 6, 3, 1, 2, 4], 1),
 ([5, 7, 2, 1, 3, 4, 6], 1), ([6, 3, 5, 1, 2, 4], 1),
 ([6, 4, 3, 1, 2, 5], 1), ([6, 5, 2, 1, 3, 4], 1),
 ([7, 3, 4, 1, 2, 5, 6], 1), ([7, 4, 2, 1, 3, 5, 6], 1)]
```

sage.libs.lrcalc.lrcalc. **skew** (*outer*, *inner*, *maxrows=0*)

Compute the Schur expansion of a skew Schur function.

Return a linear combination of partitions representing the Schur function of the skew Young diagram outer / inner , consisting of boxes in the partition outer that are not in inner .

INPUT:

- outer – a partition.
- inner – a partition.
- maxrows – an integer or None .

If maxrows is specified, then only partitions with at most this number of rows are included in the result.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import skew
sage: sorted(skew([2,1],[1]).items())
[([1, 1], 1), ([2], 1)]
```

sage.libs.lrcalc.lrcalc. **test_iterable_to_vector** (*it*)

A wrapper function for the cdef function iterable_to_vector and vector_to_list , to test that they are working correctly.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import test_iterable_to_vector
sage: x = test_iterable_to_vector([3,2,1]); x
[3, 2, 1]
```

sage.libs.lrcalc.lrcalc. **test_skewtab_to_SkewTableau** (*outer*, *inner*)

A wrapper function for the cdef function skewtab_to_SkewTableau for testing purposes.

It constructs the first LR skew tableau of shape outer/inner as an lrcalc skewtab , and converts it to a SkewTableau .

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import test_skewtab_to_SkewTableau
sage: test_skewtab_to_SkewTableau([3,2,1],[])
[[1, 1, 1], [2, 2], [3]]
sage: test_skewtab_to_SkewTableau([4,3,2,1],[1,1]).pp()
.  1  1  1
.  2  2
1  3
2
```

CHAPTER

TWENTYSEVEN

SAGE CLASS FOR PARI'S GEN TYPE

See the `PariInstance` class for documentation and examples.

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 (trac ticket #9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (trac ticket #11611, trac ticket #11854, trac ticket #11952)
- Peter Bruin (2013-11-17): move PariInstance to a separate file (trac ticket #15185)
- Jeroen Demeyer (2014-02-09): upgrade to PARI 2.7 (trac ticket #15767)
- Martin von Gagern (2014-12-17): Added some Galois functions (trac ticket #17519)
- Jeroen Demeyer (2015-01-12): upgrade to PARI 2.8 (trac ticket #16997)
- Jeroen Demeyer (2015-03-17): automatically generate methods from `pari.desc` (trac ticket #17631 and trac ticket #17860)
- Kiran Kedlaya (2016-03-23): implement infinity type

TESTS:

Before trac ticket #15654, this used to take a very long time. Now it takes much less than a second:

```
sage: pari.allocatemem(200000)
PARI stack size set to 200000 bytes, maximum size set to ...
sage: x = polygen(ZpFM(3,10))
sage: pol = ((x-1)^50 + x)
sage: pari(pol).poldisc()
2*3 + 3^4 + 2*3^6 + 3^7 + 2*3^8 + 2*3^9 + O(3^10)
```

class `sage.libs.pari.gen.`**`gen`**

Bases: *`sage.libs.pari.gen.gen_auto`*

Cython extension class that models the PARI GEN type.

`Col` (*x*, *n=0*)

Transform the object x into a column vector with minimal size $|n|$.

INPUT:

- `x` – gen
- `n` – Make the column vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI column vector (type `t_COL`)

EXAMPLES:

```
sage: pari(1.5).Col()
[1.50000000000000]~
sage: pari([1,2,3,4]).Col()
[1, 2, 3, 4]~
sage: pari('[1,2; 3,4]').Col()
[[1, 2], [3, 4]]~
sage: pari('"Sage"').Col()
["S", "a", "g", "e"]~
sage: pari('x + 3*x^3').Col()
[3, 0, 1, 0]~
sage: pari('x + 3*x^3 + O(x^5)').Col()
[1, 0, 3, 0]~
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Col(2)
[1, 2, 3, 4]~
sage: pari([1,2,3,4]).Col(-2)
[1, 2, 3, 4]~
sage: pari([1,2,3,4]).Col(6)
[1, 2, 3, 4, 0, 0]~
sage: pari([1,2,3,4]).Col(-6)
[0, 0, 1, 2, 3, 4]~
```

See also `Vec()` (create a row vector) for more examples and `Colrev()` (create a column in reversed order).

Colrev (*x, n=0*)

Transform the object x into a column vector with minimal size $|n|$. The order of the resulting vector is reversed compared to `Col()`.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, prepend zeros; if $n < 0$, append zeros.

OUTPUT:

A PARI column vector (type `t_COL`)

EXAMPLES:

```
sage: pari(1.5).Colrev()
[1.50000000000000]~
sage: pari([1,2,3,4]).Colrev()
[4, 3, 2, 1]~
sage: pari('[1,2; 3,4]').Colrev()
[[3, 4], [1, 2]]~
sage: pari('x + 3*x^3').Colrev()
[0, 1, 0, 3]~
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Colrev(2)
[4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(-2)
[4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(6)
[0, 0, 4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(-6)
[4, 3, 2, 1, 0, 0]~
```

Ser (*f*, *v=-1*, *precision=-1*)

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:

```
sage: pari(2).Ser()
2 + O(x^16)
sage: pari(Mod(0, 7)).Ser()
Mod(0, 7)*x^15 + O(x^16)

sage: x = pari([1, 2, 3, 4, 5])
sage: x.Ser()
1 + 2*x + 3*x^2 + 4*x^3 + 5*x^4 + O(x^16)
sage: f = x.Ser('v'); print(f)
1 + 2*v + 3*v^2 + 4*v^3 + 5*v^4 + O(v^16)
sage: 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 + O(v^16)

sage: pari('x^5').Ser(precision=20)
x^5 + O(x^25)
```

```
sage: pari('1/x').Ser(precision=1)
x^-1 + O(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

EXAMPLES:

```
sage: pari([1,2,['abc',1]]).Str()
"[1, 2, [abc, 1]]"
sage: pari([1,1, 1.54]).Str()
"[1, 1, 1.54000000000000]"
sage: pari(1).Str()       # 1 is automatically converted to string rep
"1"
sage: x = pari('x')       # PARI variable "x"
sage: x.Str()             # is converted to string rep.
"x"
sage: x.Str().type()
't_STR'
```

Strexpand (*x*)

Concatenate the entries of the vector x into a single string, then perform tilde expansion and environment variable expansion similar to shells.

INPUT:

- `x` – PARI gen. Either a vector or an element which is then treated like $[x]$.

OUTPUT:

- PARI string (type `t_STR`)

EXAMPLES:

```
sage: pari('"~/subdir"').Strexpand()      # random
"/home/johndoe/subdir"
sage: pari('"$SAGE_LOCAL"').Strexpand()   # random
"/usr/local/sage/local"
```

TESTS:

```
sage: a = pari('"$HOME"')
sage: a.Strexpand() != a
True
```

Strtex (*x*)

Strtex(x): Translates the vector x of PARI gens to TeX format and returns the resulting concatenated strings as a PARI t_STR.

INPUT:

- `x` – PARI gen. Either a vector or an element which is then treated like $[x]$.

OUTPUT:

- PARI string (type `t_STR`)

EXAMPLES:

```
sage: v=pari('x^2')
sage: v.Strtex()
"x^2"
sage: v=pari(['1/x^2','x'])
sage: v.Strtex()
"\\frac{1}{x^2}x"
sage: v=pari(['1 + 1/x + 1/(y+1)','x-1'])
sage: v.Strtex()
"\\frac{ \\left(y\n + 2\\right) \\*x\n + \\left(y\n + 1\\right) }{ \\left(y\n␣
↪+ 1\\right) \\*x}x\n - 1"
```

Vec (*x*, *n=0*)

Transform the object x into a vector with minimal size $|n|$.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI vector (type `t_VEC`)

EXAMPLES:

```
sage: pari(1).Vec()
[1]
sage: pari('x^3').Vec()
[1, 0, 0, 0]
sage: pari('x^3 + 3*x - 2').Vec()
[1, 0, 3, -2]
sage: pari([1,2,3]).Vec()
[1, 2, 3]
sage: pari('[1, 2; 3, 4]').Vec()
[[1, 3]~, [2, 4]~]
sage: pari('"Sage"').Vec()
["S", "a", "g", "e"]
sage: pari('2*x^2 + 3*x^3 + O(x^5)').Vec()
[2, 3, 0]
sage: pari('2*x^-2 + 3*x^3 + O(x^5)').Vec()
[2, 0, 0, 0, 0, 3, 0]
```

Note the different term ordering for polynomials and series:

```
sage: pari('1 + x + 3*x^3 + O(x^5)').Vec()
[1, 1, 0, 3, 0]
sage: pari('1 + x + 3*x^3').Vec()
[3, 0, 1, 1]
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Vec(2)
[1, 2, 3, 4]
sage: pari([1,2,3,4]).Vec(-2)
[1, 2, 3, 4]
sage: pari([1,2,3,4]).Vec(6)
```

```
[1, 2, 3, 4, 0, 0]
sage: pari([1,2,3,4]).Vec(-6)
[0, 0, 1, 2, 3, 4]
```

See also *Col()* (create a column vector) and *Vecrev()* (create a vector in reversed order).

Vecrev (*x*, *n=0*)

Transform the object x into a vector with minimal size $|n|$. The order of the resulting vector is reversed compared to *Vec()* .

INPUT:

- x – gen
- n – Make the vector of minimal length $|n|$. If $n > 0$, prepend zeros; if $n < 0$, append zeros.

OUTPUT:

A PARI vector (type t_VEC)

EXAMPLES:

```
sage: pari(1).Vecrev()
[1]
sage: pari('x^3').Vecrev()
[0, 0, 0, 1]
sage: pari('x^3 + 3*x - 2').Vecrev()
[-2, 3, 0, 1]
sage: pari([1, 2, 3]).Vecrev()
[3, 2, 1]
sage: pari('Col([1, 2, 3])').Vecrev()
[3, 2, 1]
sage: pari('[1, 2; 3, 4]').Vecrev()
[[2, 4]~, [1, 3]~]
sage: pari('"Sage"').Vecrev()
["e", "g", "a", "S"]
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Vecrev(2)
[4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(-2)
[4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(6)
[0, 0, 4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(-6)
[4, 3, 2, 1, 0, 0]
```

Vecsmall (*x*, *n=0*)

Transform the object x into a t_VECSMALL with minimal size $|n|$.

INPUT:

- x – gen
- n – Make the vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI vector of small integers (type t_VECSMALL)

EXAMPLES:

```
sage: pari([1,2,3]).Vecsmall()
Vecsmall([1, 2, 3])
sage: pari('"Sage"').Vecsmall()
Vecsmall([83, 97, 103, 101])
sage: pari(1234).Vecsmall()
Vecsmall([1234])
sage: pari('x^2 + 2*x + 3').Vecsmall()
Vecsmall([1, 2, 3])
```

We demonstate the n argument:

```
sage: pari([1,2,3]).Vecsmall(2)
Vecsmall([1, 2, 3])
sage: pari([1,2,3]).Vecsmall(-2)
Vecsmall([1, 2, 3])
sage: pari([1,2,3]).Vecsmall(6)
Vecsmall([1, 2, 3, 0, 0, 0])
sage: pari([1,2,3]).Vecsmall(-6)
Vecsmall([0, 0, 0, 1, 2, 3])
```

Zn_issquare (*n*)

Return `True` if `self` is a square modulo n, `False` if not.

INPUT:

- `self` – integer
- `n` – integer or factorisation matrix

EXAMPLES:

```
sage: pari(3).Zn_issquare(4)
False
sage: pari(4).Zn_issquare(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:

```
sage: pari(3).Zn_sqrt(4)
Traceback (most recent call last):
...
ValueError: 3 is not a square modulo 4
sage: pari(4).Zn_sqrt(30.factor())
22
```

bernfrac (*x*)

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:

```
sage: pari(18).bernfrac()
43867/798
sage: [pari(n).bernfrac() for n in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
```

bernreal (*x*, *precision=0*)

The Bernoulli number B_x, as for the function bernfrac, but B_x is returned as a real number (with the current precision).

EXAMPLES:

```
sage: pari(18).bernreal()
54.9711779448622
sage: pari(18).bernreal(precision=192).sage()
54.9711779448621553884711779448621553884711779448621553885
```

bernvec (*x*)

Creates a vector containing, as rational numbers, the Bernoulli numbers $B_0, B_2, \ldots, B_{2x}$. This routine is obsolete. Use bernfrac instead each time you need a Bernoulli number in exact form.

Note: this routine is implemented using repeated independent calls to bernfrac, which is faster than the standard recursion in exact arithmetic.

EXAMPLES:

```
sage: pari(8).bernvec()
doctest:...: DeprecationWarning: bernvec() is deprecated, use repeated calls␣
↪to bernfrac() instead
See http://trac.sagemath.org/15767 for details.
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
sage: [pari(2*n).bernfrac() for n in range(9)]
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
```

besselk (*nu*, *x*, *flag=None*, *precision=0*)

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:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).besselk(3)
0.0455907718407551 + 0.0289192946582081*I
```

```
sage: pari(2+i).besselk(-3)
-4.34870874986752 - 5.38744882697109*I
```

```
sage: pari(2+i).besselk(300)
3.74224603319728 E-132 + 2.49071062641525 E-134*I
sage: pari(2+i).besselk(300, flag=1)
```

```
doctest:...: DeprecationWarning: The flag argument to besselk() is deprecated␣
↪and not used anymore
See http://trac.sagemath.org/20219 for details.
3.74224603319728 E-132 + 2.49071062641525 E-134*I
```

bezout (*x*, *y*)

bezoutres (**args*, ***kwds*)

Deprecated: Use `polresultantext()` instead. See trac ticket #18203 for details.

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:

```
sage: K.<i> = QuadraticField(-1)
sage: J = pari(K).idealstar(K.ideal(4*i + 2))
sage: 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:

```
sage: K.<i> = QuadraticField(-1)
sage: J = pari(K).idealstar(K.ideal(4*i + 2), 2)
sage: J.bid_get_gen()
[7, [-2, -1]~]
```

We get an exception if we do not supply `flag = 2` to `idealstar`:

```
sage: J = pari(K).idealstar(K.ideal(3))
sage: J.bid_get_gen()
Traceback (most recent call last):
...
PariError: missing bid generators. Use idealstar(,,2)
```

bittest (*x*, *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:

```
sage: x = pari(6)
sage: x.bittest(0)
False
sage: x.bittest(1)
True
```

```
sage: x.bittest(2)
True
sage: x.bittest(3)
False
sage: pari(-3).bittest(0)
True
sage: pari(-3).bittest(1)
False
sage: [pari(-3).bittest(n) for n in range(10)]
[True, False, True, True, 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:

```
sage: K.<a> = QuadraticField(-65)
sage: K.pari_bnf().bnf_get_cyc()
[4, 2]
```

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:

```
sage: K.<a> = QuadraticField(-65)
sage: G = K.pari_bnf().bnf_get_gen(); G
[[3, 2; 0, 1], [2, 1; 0, 1]]
sage: [K.ideal(J) for J in G]
[Fractional ideal (3, a + 2), Fractional ideal (2, a + 1)]
```

bnf_get_no ( )

Returns the class number of `self` , a "big number field" (`bnf`).

EXAMPLES:

```
sage: K.<a> = QuadraticField(-65)
sage: K.pari_bnf().bnf_get_no()
8
```

bnf_get_reg ( )

Returns the regulator of this number field.

NOTE: `self` must be a "big number field" (`bnf`).

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: K.pari_bnf().bnf_get_reg()
2.66089858019037...
```

bnfunit ( )

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()`.

EXAMPLES:

```
sage: f = pari('x^3 + 17*x + 3')
sage: f.change_variable_name("y")
y^3 + 17*y + 3
sage: f = pari('1 + 2*y + O(y^10)')
sage: f.change_variable_name("q")
1 + 2*q + O(q^10)
sage: f.change_variable_name("y") is f
True
```

In PARI, `I` refers to the square root of -1, so it cannot be used as variable name. Note the difference with `subst()`:

```
sage: f = pari('x^2 + 1')
sage: f.change_variable_name("I")
Traceback (most recent call last):
...
PariError: I already exists with incompatible valence
sage: f.subst("x", "I")
0
```

debug (*depth=-1*)
Show the internal structure of self (like the `\x` command in gp).

EXAMPLE:

```
sage: pari('[1/2, 1.0*I]').debug()  # random addresses
[&=0000000004c5f010] VEC(lg=3):2200000000000003 0000000004c5eff8␣
↪0000000004c5efb0
  1st component = [&=0000000004c5eff8] FRAC(lg=3):0800000000000003␣
↪0000000004c5efe0 0000000004c5efc8
    num = [&=0000000004c5efe0] INT(lg=3):0200000000000003 (+,lgefint=3):
↪4000000000000003 0000000000000001
    den = [&=0000000004c5efc8] INT(lg=3):0200000000000003 (+,lgefint=3):
↪4000000000000003 0000000000000002
  2nd component = [&=0000000004c5efb0] COMPLEX(lg=3):0c00000000000003␣
↪00007fae8a2eb840 0000000004c5ef90
    real = gen_0
    imag = [&=0000000004c5ef90] REAL(lg=4):0400000000000004 (+,expo=0):
↪6000000000000000 8000000000000000 0000000000000000
```

disc ()
Return the discriminant of this object.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.disc()
-161051
sage: _.factor()
[-1, 1; 11, 5]
```

eint1 (*x*, *n=0*, *precision=0*)
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:

elementval (**args*, ***kwds*)
Deprecated: Use `nfeltval()` instead. See trac ticket #20219 for details.

ellan (*n*, *python_ints=False*)
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:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellan(3)
[1, -2, -1]
sage: e.ellan(20)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]
sage: e.ellan(-1)
[]
sage: v = e.ellan(10, python_ints=True); v
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

ellaplist (*n*, *python_ints=False*)
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!)

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: v = e.ellaplist(10); v
[-2, -1, 1, -2]
sage: type(v)
<type 'sage.libs.pari.gen.gen'>
sage: v.type()
't_VEC'
sage: e.ellan(10)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
sage: v = e.ellaplist(10, python_ints=True); v
[-2, -1, 1, -2]
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

TESTS:

```
sage: v = e.ellaplist(1)
sage: v, type(v)
([], <type 'sage.libs.pari.gen.gen'>)
sage: v = e.ellaplist(1, python_ints=True)
sage: v, type(v)
([], <type 'list'>)
```

ellbil (**args*, ***kwds*)

Deprecated: Use `ellheight()` instead. See trac ticket #18203 for details.

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:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.ellisoncurve([1,0])
True
sage: e.ellisoncurve([1,1])
False
sage: e.ellisoncurve([1,0.00000000000000001])
False
sage: e.ellisoncurve([1,0.000000000000000001])
True
sage: 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:

```
sage: e = pari([1,2,3,4,5]).ellinit()
sage: F, ch = e.ellminimalmodel()
sage: F[:5]
[1, -1, 0, 4, 3]
sage: ch
[1, -1, 0, -1]
sage: e.ellchangecurve(ch)[:5]
[1, -1, 0, 4, 3]
```

ellpow (**args*, ***kwds*)

Deprecated: Use `ellmul()` instead. See trac ticket #18203 for details.

elltors (*flag=None*)

Return information about the torsion subgroup of the given elliptic curve.

INPUT:

- `e` - elliptic curve over $\mathbf{Q}$

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 non-increasing order
- `gen` - vector giving points on e generating these cyclic groups

EXAMPLES:

```
sage: e = pari([1,0,1,-19,26]).ellinit()
sage: e.elltors()
[12, [6, 2], [[1, 2], [3, -2]]]
```

ellwp (*z='z'*, *n=20*, *flag=0*, *precision=0*)

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):

```
sage: E = pari([0,-1,1,-10,-20]).ellinit()
```

Compute P(1):

```
sage: E.ellwp(1)
13.9658695257485
```

Compute P(1+i), where i = sqrt(-1):

```
sage: C.<i> = ComplexField()
sage: E.ellwp(pari(1+i))
-1.11510682565555 + 2.33419052307470*I
sage: E.ellwp(1+i)
-1.11510682565555 + 2.33419052307470*I
```

The series expansion, to the default $O(z^20)$ precision:

```
sage: E.ellwp()
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8 + 1202285717/
↪928746000*z^10 + 2403461/2806650*z^12 + 30211462703/43418875500*z^14 +␣
↪3539374016033/7723451736000*z^16 + 413306031683977/1289540602350000*z^18 +␣
↪O(z^20)
```

Compute the series for wp to lower precision:

```
sage: E.ellwp(n=4)
z^-2 + 31/15*z^2 + O(z^4)
```

Next we use the version where the input is generators for a lattice:

```
sage: pari([1.2692, 0.63 + 1.45*i]).ellwp(1)
13.9656146936689 + 0.000644829272810...*I
```

With flag=1, compute the pair P(z) and P'(z):

```
sage: E.ellwp(1, flag=1)
[13.9658695257485, 50.5619300880073]
```

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 `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:

```
sage: f = pari('x^2 + 1')
sage: f.type()
't_POL'
sage: f.eval(I)
```

```
0
sage: f.eval(x=2)
5
sage: (1/f).eval(x=1)
1/2
```

The notation `f(x)` is an alternative for `f.eval(x)`:

```
sage: f(3) == f.eval(3)
True
```

Evaluating power series:

```
sage: f = pari('1 + x + x^3 + O(x^7)')
sage: f(2*pari('y')^2)
1 + 2*y^2 + 8*y^6 + O(y^14)
```

Substituting zero is sometimes possible, and trying to do so in illegal cases can raise various errors:

```
sage: pari('1 + O(x^3)').eval(0)
1
sage: pari('1/x').eval(0)
Traceback (most recent call last):
...
PariError: impossible inverse in gdiv: 0
sage: pari('1/x + O(x^2)').eval(0)
Traceback (most recent call last):
...
ZeroDivisionError: substituting 0 in Laurent series with negative valuation
sage: pari('1/x + O(x^2)').eval(pari('O(x^3)'))
Traceback (most recent call last):
...
PariError: impossible inverse in gdiv: O(x^3)
sage: pari('O(x^0)').eval(0)
Traceback (most recent call last):
...
PariError: domain error in polcoeff: t_SER = O(x^0)
```

Evaluating multivariate polynomials:

```
sage: f = pari('y^2 + x^3')
sage: f(1)     # Dangerous, depends on PARI variable ordering
y^2 + 1
sage: f(x=1)   # Safe
y^2 + 1
sage: f(y=1)
x^3 + 1
sage: f(1, 2)
Traceback (most recent call last):
...
TypeError: evaluating PARI t_POL takes exactly 1 argument (2 given)
sage: f(y='x', x='2*y')
x^2 + 8*y^3
sage: f()
x^3 + y^2
```

It's not an error to substitute variables which do not appear:

```
sage: f.eval(z=37)
x^3 + y^2
sage: pari(42).eval(t=0)
42
```

We can define and evaluate closures as follows:

```
sage: T = pari('n -> n + 2')
sage: T.type()
't_CLOSURE'
sage: T.eval(3)
5

sage: T = pari('() -> 42')
sage: T()
42

sage: pr = pari('s -> print(s)')
sage: pr.eval('"hello world"')
hello world

sage: f = pari('myfunc(x,y) = x*y')
sage: f.eval(5, 6)
30
```

Default arguments work, missing arguments are treated as zero (like in GP):

```
sage: f = pari("(x, y, z=1.0) -> [x, y, z]")
sage: f(1, 2, 3)
[1, 2, 3]
sage: f(1, 2)
[1, 2, 1.00000000000000]
sage: f(1)
[1, 0, 1.00000000000000]
sage: f()
[0, 0, 1.00000000000000]
```

Variadic closures are supported as well (trac ticket #18623):

```
sage: f = pari("(v[..])->length(v)")
sage: f('a', f)
2
sage: g = pari("(x,y,z[..])->[x,y,z]")
sage: 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:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: nf = K._pari_()
sage: nf
[y^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-38 + 1.00000000000000*I]), [1, 1.
↪00000000000000; 1, -1.00000000000000], [1, 1; 1, -1], [2, 0; 0, -2], [2, 0;␣
↪0, 2], [1, 0; 0, -1], [1, [0, -1; 1, 0]], []], [0.E-38 + 1.
↪00000000000000*I], [1, y], [1, 0; 0, 1], [1, 0, 0, -1; 0, 1, 1, 0]]
sage: nf(y='x')
[x^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-38 + 1.00000000000000*I]), [1, 1.
↪00000000000000; 1, -1.00000000000000], [1, 1; 1, -1], [2, 0; 0, -2], [2, 0;␣
↪0, 2], [1, 0; 0, -1], [1, [0, -1; 1, 0]], []], [0.E-38 + 1.
↪00000000000000*I], [1, x], [1, 0; 0, 1], [1, 0, 0, -1; 0, 1, 1, 0]]
```

factor (*limit=-1*, *proof=None*)

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 Sage.

EXAMPLES:

```
sage: 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]
sage: 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]
sage: 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]
sage: 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:

```
sage: pari(next_prime(10^50)*next_prime(10^60)*next_prime(10^4)).factor(10^5)
[10007, 1;␣
↪10000000000000000000000000000000000000000000000015100000000070000000000000000000000000000000
↪1]
```

Setting a limit is invalid when factoring polynomials:

```
sage: pari('x^11 + 1').factor(limit=17)
Traceback (most recent call last):
...
PariError: incorrect type in boundfact (t_POL)
```

PARI doesn't have an algorithm for factoring multivariate polynomials:

```
sage: pari('x^3 - y^3').factor()
Traceback (most recent call last):
...
PariError: sorry, factor for general polynomials is not yet implemented
```

TESTS:

```
sage: pari(2^1000+1).factor(limit=0)
doctest:...: DeprecationWarning: factor(..., lim=0) is deprecated, use an␣
↪explicit limit instead
See http://trac.sagemath.org/20205 for details.
[257, 1; 1601, 1; 25601, 1; 76001, 1; 133842787352016..., 1]
```

factorpadic (*p*, *r=20*)

p-adic factorization of the polynomial `pol` to precision `r` .

EXAMPLES:

```
sage: x = polygen(QQ)
sage: pol = (x^2 - 1)^2
sage: pari(pol).factorpadic(5)
[(1 + O(5^20))*x + (1 + O(5^20)), 2; (1 + O(5^20))*x + (4 + 4*5 + 4*5^2 + 4*5^
↪3 + 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 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^18 + 4*5^19 + O(5^
↪20)), 2]
sage: pari(pol).factorpadic(5,3)
[(1 + O(5^3))*x + (1 + O(5^3)), 2; (1 + O(5^3))*x + (4 + 4*5 + 4*5^2 + O(5^
↪3)), 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:

```
sage: x = polygen(GF(3))
sage: k.<a> = GF(9, modulus=x^2+1)
sage: b = pari(a).ffprimroot()
sage: b  # random
a + 1
sage: b.fforder()
8
```

fibonacci ( )

Return the Fibonacci number of index x.

EXAMPLES:

```
sage: pari(18).fibonacci()
2584
sage: [pari(n).fibonacci() for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

galoissubfields (*flag=0*, *v=-1*)

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:

```
sage: G = pari(x^6 + 108).galoisinit()
sage: G.galoissubfields(flag=1)
[x, x^2 + 972, x^3 + 54, x^3 + 864, x^3 - 54, x^6 + 108]
sage: G = pari(x^4 + 1).galoisinit()
sage: G.galoissubfields(flag=2, v='z')[3]
[x^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - z*x - 1, x^2 + z*x - 1]]
```

gequal (*a*, *b*)

Check whether a and b are equal using PARI's `gequal`.

EXAMPLES:

```
sage: a = pari(1); b = pari(1.0); c = pari('"some_string"')
sage: a.gequal(a)
True
sage: b.gequal(b)
True
sage: c.gequal(c)
True
sage: a.gequal(b)
True
sage: a.gequal(c)
False
```

WARNING: this relation is not transitive:

```
sage: a = pari('[0]'); b = pari(0); c = pari('[0,0]')
sage: a.gequal(b)
True
sage: b.gequal(c)
True
sage: a.gequal(c)
False
```

gequal0 (*a*)

Check whether a is equal to zero.

EXAMPLES:

```
sage: pari(0).gequal0()
True
sage: pari(1).gequal0()
False
sage: pari(1e-100).gequal0()
False
sage: pari("0.0 + 0.0*I").gequal0()
True
sage: pari(GF(3^20,'t')(0)).gequal0()
True
```

gequal_long (*a*, *b*)

Check whether a is equal to the `long int` b using PARI's `gequalsg`.

EXAMPLES:

```
sage: a = pari(1); b = pari(2.0); c = pari('3*matid(3)')
sage: a.gequal_long(1)
```

```
True
sage: a.gequal_long(-1)
False
sage: a.gequal_long(0)
False
sage: b.gequal_long(2)
True
sage: b.gequal_long(-2)
False
sage: c.gequal_long(3)
True
sage: c.gequal_long(-3)
False
```

getattr (*attr*)

Return the PARI attribute with the given name.

EXAMPLES:

```
sage: K = pari("nfinit(x^2 - x - 1)")
sage: K.getattr("pol")
x^2 - x - 1
sage: K.getattr("disc")
5

sage: K.getattr("reg")
Traceback (most recent call last):
...
PariError: _.reg: incorrect type in reg (t_VEC)
sage: K.getattr("zzz")
Traceback (most recent call last):
...
PariError: not a function in function call
```

idealintersection (**args*, ***kwds*)

Deprecated: Use `idealintersect()` instead. See trac ticket #20219 for details.

ispower (*k=None*)

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:

```
sage: pari(9).ispower()
(2, 3)
sage: pari(17).ispower()
(1, 17)
sage: pari(17).ispower(2)
(False, None)
sage: pari(17).ispower(1)
```

```
(1, 17)
sage: pari(2).ispower()
(1, 2)
```

isprime (*flag=0*)

isprime(x, flag=0): Returns True if x is a PROVEN prime number, and False otherwise.

INPUT:

- `flag` - int 0 (default): use a combination of algorithms. 1: certify primality using the Pocklington-Lehmer Test. 2: certify primality using the APRCL test.

OUTPUT:

- `bool` - True or False

EXAMPLES:

```
sage: pari(9).isprime()
False
sage: pari(17).isprime()
True
sage: n = pari(561)     # smallest Carmichael number
sage: n.isprime()       # not just a pseudo-primality test!
False
sage: n.isprime(1)
False
sage: n.isprime(2)
False
sage: n = pari(2^31-1)
sage: n.isprime(1)
(True, [2, 3, 1; 3, 5, 1; 7, 3, 1; 11, 3, 1; 31, 2, 1; 151, 3, 1; 331, 3, 1])
```

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.

EXAMPLES:

```
sage: pari(9).isprimepower()
(2, 3)
sage: pari(17).isprimepower()
(1, 17)
sage: pari(18).isprimepower()
(0, 18)
sage: pari(3^12345).isprimepower()
(12345, 3)
```

ispseudoprime (*flag=0*)

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:

```
sage: pari(9).ispseudoprime()
False
sage: pari(17).ispseudoprime()
True
sage: n = pari(561)     # smallest Carmichael number
sage: 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:

```
sage: pari(3^12345).ispseudoprimepower()
(12345, 3)
sage: p = pari(2^1500 + 1465)         # next_prime(2^1500)
sage: (p^11).ispseudoprimepower()[0]  # very fast
11
```

issquare (*x*, *find_root=False*)

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:

```
sage: pari(10).issquarefree()
True
sage: pari(20).issquarefree()
False
```

j ( )

Return the j-invariant of this object.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.j()
-122023936/161051
sage: _.factor()
[-1, 1; 2, 12; 11, -5; 31, 3]
```

lift_centered (*x*, *v=None*)

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 non-negative; 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 list of PARI gens.

EXAMPLES:

A PARI vector becomes a Sage list:

```
sage: L = pari("vector(10,i,i^2)").list()
sage: L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
sage: type(L)
<type 'list'>
sage: type(L[0])
<type 'sage.libs.pari.gen.gen'>
```

For polynomials, list() behaves as for ordinary Sage polynomials:

```
sage: 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:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: s = x^2 + O(x^8)
sage: s.list()
[1]
sage: pari(s).list()
[1, 0, 0, 0, 0, 0]
sage: s = x^-2 + O(x^0)
sage: s.list()
[1]
sage: pari(s).list()
[1, 0]
```

For matrices, we get a list of columns:

```
sage: M = matrix(ZZ,3,2,[1,4,2,5,3,6]); M
[1 4]
[2 5]
[3 6]
sage: pari(M).list()
[[1, 2, 3]~, [4, 5, 6]~]
```

For “scalar” types, we get a 1-element list containing `self` :

```
sage: pari("42").list()
[42]
```

list_str ( )

Return str that might correctly evaluate to a Python-list.

TESTS:

```
sage: pari.primes(5).list_str()
doctest:...: DeprecationWarning: the method list_str() is deprecated
See http://trac.sagemath.org/20219 for details.
[2, 3, 5, 7, 11]
```

lllgram ( )

lllgramint ( )

log_gamma (*x*, *precision=0*)

Principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with non-positive 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+O(x^2))
%1 = -0.57721566490153286060651209008240243104*x + O(x^2)
? lngamma(x+O(x^2))
 *** at top-level: lngamma(x+O(x^2))
 *** ^-----------------
 *** lngamma: domain error in lngamma: valuation != 0
? lngamma(-1+x+O(x^2))
 *** lngamma: Warning: normalizing a series with 0 leading term.
 *** at top-level: lngamma(-1+x+O(x^2))
 *** ^--------------------
 *** lngamma: domain error in intformal: residue(series, pole) != 0
```

matkerint (*flag=0*)

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:

```
sage: pari('[2,1;2,1]').matker()
[-1/2; 1]
sage: pari('[2,1;2,1]').matkerint()
[1; -2]
sage: pari('[2,1;2,1]').matkerint(1)
doctest:...: DeprecationWarning: The flag argument to matkerint() is
↪deprecated by PARI
```

```
See http://trac.sagemath.org/18203 for details.
[1; -2]
```

mattranspose ( )

Transpose of the matrix self.

EXAMPLES:

```
sage: 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:

```
sage: pari('[1,2,3]').mattranspose()
[1; 2; 3]
sage: pari('[1,2,3]~').mattranspose()
Mat([1, 2, 3])
```

mod ( )

Given an INTMOD or POLMOD `Mod(a,m)` , return the modulus m.

EXAMPLES:

```
sage: pari(4).Mod(5).mod()
5
sage: pari("Mod(x, x*y)").mod()
y*x
sage: pari("[Mod(4,5)]").mod()
Traceback (most recent call last):
...
TypeError: Not an INTMOD or POLMOD in mod()
```

multiplicative_order (*x*, *o=None*)

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 non-zero 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.

EXAMPLES:

```
sage: pari('matrix(19,8)').ncols()
8
```

nextprime (*add_one=0*)

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:

```
sage: pari(1).nextprime()
2
sage: pari(2).nextprime()
2
sage: pari(2).nextprime(add_one = 1)
3
```

```
sage: 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:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(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:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(K).nf_get_pol()
y^4 - 4*y^2 + 1
sage: bnr = pari("K = bnfinit(x^4 - 4*x^2 + 1); bnrinit(K, 2*x)")
sage: bnr.nf_get_pol()
x^4 - 4*x^2 + 1
```

For relative number fields, this returns the relative polynomial. However, beware that `pari(L)` returns an absolute number field:

```
sage: L.<b> = K.extension(x^2 - 5)
sage: pari(L).nf_get_pol()        # Absolute
y^8 - 28*y^6 + 208*y^4 - 408*y^2 + 36
sage: L.pari_rnf().nf_get_pol()   # Relative
x^2 - 5
```

TESTS:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: K.pari_nf().nf_get_pol()
y^4 - 4*y^2 + 1
sage: K.pari_bnf().nf_get_pol()
y^4 - 4*y^2 + 1
```

An error is raised for invalid input:

```
sage: 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:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: s = K.pari_nf().nf_get_sign(); s
[4, 0]
sage: type(s); type(s[0])
<type 'list'>
<type 'int'>
sage: CyclotomicField(15).pari_nf().nf_get_sign()
[0, 4]
```

nf_get_zk ( )

Returns a vector with a $\mathbf{Z}$-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() .

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(K).nf_get_zk()
[1, y, y^3 - 4*y, y^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:

```
sage: x = polygen(QQ)
sage: K = NumberField(x^2 + 5, 'a')
```

We can substitute in a PARI `nf` structure:

```
sage: Kpari = K.pari_nf()
sage: Kpari.nf_get_pol()
y^2 + 5
sage: Lpari = Kpari.nf_subst('a')
sage: Lpari.nf_get_pol()
a^2 + 5
```

We can also substitute in a PARI `bnf` structure:

```
sage: Kpari = K.pari_bnf()
sage: Kpari.nf_get_pol()
y^2 + 5
sage: Kpari.bnf_get_cyc()   # Structure of class group
[2]
sage: Lpari = Kpari.nf_subst('a')
sage: Lpari.nf_get_pol()
a^2 + 5
sage: Lpari.bnf_get_cyc()   # We still have a bnf after substituting
[2]
```

nfbasis (*flag=0*, *fa=None*)

Integral basis of the field $\mathbf{Q}[a]$, 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` .

Note: In earlier versions of Sage, other bits in `flag` were defined but these are now simply ignored.

EXAMPLES:

```
sage: 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:

```
sage: p = next_prime(10^10); q = next_prime(p)
sage: x = polygen(QQ); f = x^2 + p^2*q
sage: pari(f).nfbasis(1)   # Wrong result
[1, x]
sage: pari(f).nfbasis()    # Correct result
[1, 1/10000000019*x]
sage: pari(f).nfbasis(fa=10^6)   # Check primes up to 10^6: wrong result
[1, x]
sage: pari(f).nfbasis(fa="[2,2; %s,2]"%p)    # Correct result and faster
[1, 1/10000000019*x]
sage: pari(f).nfbasis(fa=[2,p])              # Equivalent with the above
[1, 1/10000000019*x]
```

nfbasis_d (*flag=0*, *fa=None*)

Like *nfbasis()* , but return a tuple `(B,D)` where B is the integral basis and D the discriminant.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: F._pari_()[0].nfbasis_d()
([1, y, y^2], -108)
```

```
sage: G = NumberField(x^5-11,'beta')
sage: G._pari_()[0].nfbasis_d()
([1, y, y^2, y^3, y^4], 45753125)
```

```
sage: pari([-2,0,0,1]).Polrev().nfbasis_d()
([1, x, x^2], -108)
```

nfbasistoalg_lift (*nf*, *x*)

Transforms the column vector `x` on the integral basis into a polynomial representing the algebraic number.

INPUT:

- `nf` – a number field
- `x` – a column of rational numbers of length equal to the degree of `nf` or a single rational number

OUTPUT:

- `nf.nfbasistoalg(x).lift()`

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^3 - 17)
sage: Kpari = K.pari_nf()
sage: Kpari.getattr('zk')
[1, 1/3*y^2 - 1/3*y + 1/3, y]
sage: Kpari.nfbasistoalg_lift(42)
42
sage: Kpari.nfbasistoalg_lift("[3/2, -5, 0]~")
-5/3*y^2 + 5/3*y - 1/6
sage: Kpari.getattr('zk') * pari("[3/2, -5, 0]~")
-5/3*y^2 + 5/3*y - 1/6
```

nfeltval (*x*, *p*)

Return the valuation of the number field element x at the prime p.

EXAMPLES:

```
sage: nf = pari('x^2 + 1').nfinit()
sage: p = nf.idealprimedec(5)[0]
sage: nf.nfeltval('50 - 25*x', p)
3
```

nfgenerator ( )

nrows ( )

Return the number of rows of self.

EXAMPLES:

```
sage: pari('matrix(19,8)').nrows()
19
```

omega (*precision=0*)

Return the basis for the period lattice of this elliptic curve.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.omega()
[1.26920930427955, 0.634604652139777 - 1.45881661693850*I]
```

order (**args*, ***kwds*)

Deprecated: Use `znorder()` instead. See trac ticket #20219 for details.

padicprime (*x*)

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:

```
sage: K = Qp(11,5)
sage: x = K(11^-10 + 5*11^-7 + 11^-6)
sage: y = pari(x)
sage: y.padicprime()
11
sage: y.padicprime().type()
't_INT'
```

phi (**args*, ***kwds*)

Deprecated: Use `eulerphi()` instead. See trac ticket #20219 for details.

poldegree (*var=-1*)

Return the degree of this polynomial.

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.

polisirreducible ( )

f.polisirreducible(): Returns True if f is an irreducible non-constant polynomial, or False if f is reducible or constant.

polroots (*precision=0*)

Complex roots of the given polynomial using Schonhage's method, as modified by Gourdon.

polylog (*x*, *m*, *flag=0*, *precision=0*)

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:

```
sage: pari(10).polylog(3)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,0)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,1)
0.523778453502411
sage: pari(10).polylog(3,2)
-0.400459056163451
```

pr_get_e ( )

Returns the ramification index (over $\mathbf{Q}$) of this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: pari(K).idealfactor(K.ideal(2))[0,0].pr_get_e()
2
```

```
sage: pari(K).idealfactor(K.ideal(3))[0,0].pr_get_e()
1
sage: pari(K).idealfactor(K.ideal(5))[0,0].pr_get_e()
1
```

pr_get_f ( )

Returns the residue class degree (over $\mathbf{Q}$) of this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: pari(K).idealfactor(K.ideal(2))[0,0].pr_get_f()
1
sage: pari(K).idealfactor(K.ideal(3))[0,0].pr_get_f()
2
sage: pari(K).idealfactor(K.ideal(5))[0,0].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 `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: g = pari(K).idealfactor(K.ideal(2))[0,0].pr_get_gen(); g; K(g)
[1, 1]~
i + 1
sage: g = pari(K).idealfactor(K.ideal(3))[0,0].pr_get_gen(); g; K(g)
[3, 0]~
3
sage: g = pari(K).idealfactor(K.ideal(5))[0,0].pr_get_gen(); g; K(g)
[-2, 1]~
i - 2
```

pr_get_p ( )

Returns the prime of $\mathbf{Z}$ lying below this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: F = pari(K).idealfactor(K.ideal(5)); F
[[5, [-2, 1]~, 1, 1, [2, -1; 1, 2]], 1; [5, [2, 1]~, 1, 1, [-2, -1; 1, -2]],
↪1]
sage: F[0,0].pr_get_p()
5
```

precision (*x*, *n=-1*)

Change the precision of x to be n, where n is an integer. If n is omitted, output the real precision of x.

INPUT:

- `x` - gen
- `n` - (optional) int

OUTPUT: gen

printtex (**args*, ***kwds*)

Deprecated: Use *Strtex()* instead. See trac ticket #20219 for details.

python (*locals=None*)

Return the closest Python/Sage equivalent of the given PARI object.

INPUT:

- z – PARI gen
- $locals$ – optional dictionary used in fallback cases that involve `sage_eval()`

Note: If `self` is a real (type `t_REAL`), then the result will be a RealField element of the equivalent precision; if `self` is a complex (type `t_COMPLEX`), then the result will be a ComplexField element of precision the maximal precision of the real and imaginary parts.

EXAMPLES:

```
sage: pari('389/17').python()
389/17
sage: f = pari('(2/3)*x^3 + x - 5/7 + y'); f
2/3*x^3 + x + (y - 5/7)
sage: var('x,y')
(x, y)
sage: f.python({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

You can also use *sage()*, which is an alias:

```
sage: f.sage({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

Converting a real number:

```
sage: pari.set_real_precision(70)
15
sage: a = pari('1.234').python(); a
1.234000000000000000000000000000000000000000000000000000000000000000000000
sage: a.parent()
Real Field with 256 bits of precision
sage: pari.set_real_precision(15)
70
sage: a = pari('1.234').python(); a
1.23400000000000000
sage: a.parent()
Real Field with 64 bits of precision
```

For complex numbers, the parent depends on the PARI type:

```
sage: a = pari('(3+I)').python(); a
i + 3
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: a = pari('2^31-1').python(); a
2147483647
```

```
sage: a.parent()
Integer Ring

sage: a = pari('12/34').python(); a
6/17
sage: a.parent()
Rational Field

sage: a = pari('(3+I)/2').python(); a
1/2*i + 3/2
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: z = pari(CC(1.0+2.0*I)); z
1.00000000000000 + 2.00000000000000*I
sage: a = z.python(); a
1.00000000000000000 + 2.00000000000000000*I
sage: a.parent()
Complex Field with 64 bits of precision

sage: I = sqrt(-1)
sage: a = pari(1.0 + 2.0*I).python(); a
1.00000000000000000 + 2.00000000000000000*I
sage: a.parent()
Complex Field with 64 bits of precision
```

Vectors and matrices:

```
sage: a = pari('[1,2,3,4]')
sage: a
[1, 2, 3, 4]
sage: a.type()
't_VEC'
sage: b = a.python(); b
[1, 2, 3, 4]
sage: type(b)
<type 'list'>

sage: a = pari('[1,2;3,4]')
sage: a.type()
't_MAT'
sage: b = a.python(); b
[1 2]
[3 4]
sage: b.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

sage: a = pari('Vecsmall([1,2,3,4])')
sage: a.type()
't_VECSMALL'
sage: a.python()
[1, 2, 3, 4]
```

We use the locals dictionary:

```
sage: f = pari('(2/3)*x^3 + x - 5/7 + y')
sage: x,y=var('x,y')
sage: from sage.libs.pari.gen import gentoobj
```

```
sage: gentoobj(f, {'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
sage: gentoobj(f)
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Conversion of p-adics:

```
sage: K = Qp(11,5)
sage: x = K(11^-10 + 5*11^-7 + 11^-6); x
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: y = pari(x); y
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: y.sage()
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: pari(K(11^-5)).sage()
11^-5 + O(11^0)
```

Conversion of infinities:

```
sage: pari('oo').sage()
+Infinity
sage: pari('-oo').sage()
-Infinity
```

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:

```
sage: v = pari([1,2,3,10,102,10])
sage: w = v.python_list()
sage: w
[1, 2, 3, 10, 102, 10]
sage: type(w[0])
<type 'sage.libs.pari.gen.gen'>
sage: pari("[1,2,3]").python_list()
[1, 2, 3]

sage: 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:

```
sage: v=pari([1,2,3,10,102,10]).Vecsmall()
sage: w = v.python_list_small()
sage: w
[1, 2, 3, 10, 102, 10]
```

```
sage: type(w[0])
<type 'int'>
```

qfrep (*B*, *flag=0*)

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 $2B$, 2: return a `t_VECSMALL` instead of a `t_VEC` (which is faster).

EXAMPLES:

```
sage: M = pari("[5,1,1;1,3,1;1,1,1]")
sage: M.qfrep(20)
[1, 1, 2, 2, 2, 4, 4, 3, 3, 4, 2, 4, 6, 0, 4, 6, 4, 5, 6, 4]
sage: 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]
sage: 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])
```

reverse (**args*, ***kwds*)

Deprecated: Use `polrecip()` instead. See trac ticket #20219 for details.

rnfisnorm (*T*, *flag=0*)

rnfpolred (**args*, ***kwds*)

rnfpolredabs (**args*, ***kwds*)

round (*x*, *estimate=False*)

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:

```
sage: pari('1.5').round()
2
sage: pari('1.5').round(True)
(2, -1)
sage: pari('1.5 + 2.1*I').round()
2 + 2*I
sage: pari('1.0001').round(True)
(1, -14)
sage: pari('(2.4*x^2 - 1.7)/x').round()
(2*x^2 - 2)/x
```

```
sage: pari('(2.4*x^2 - 1.7)/x').truncate()
2.40000000000000*x
```

sage (*locals=None*)

Return the closest Python/Sage equivalent of the given PARI object.

INPUT:

- z – PARI `gen`
- *locals* – optional dictionary used in fallback cases that involve `sage_eval()`

Note: If `self` is a real (type `t_REAL`), then the result will be a RealField element of the equivalent precision; if `self` is a complex (type `t_COMPLEX`), then the result will be a ComplexField element of precision the maximal precision of the real and imaginary parts.

EXAMPLES:

```
sage: pari('389/17').python()
389/17
sage: f = pari('(2/3)*x^3 + x - 5/7 + y'); f
2/3*x^3 + x + (y - 5/7)
sage: var('x,y')
(x, y)
sage: f.python({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

You can also use *sage()*, which is an alias:

```
sage: f.sage({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

Converting a real number:

```
sage: pari.set_real_precision(70)
15
sage: a = pari('1.234').python(); a
1.234000000000000000000000000000000000000000000000000000000000000000000000000
sage: a.parent()
Real Field with 256 bits of precision
sage: pari.set_real_precision(15)
70
sage: a = pari('1.234').python(); a
1.23400000000000000
sage: a.parent()
Real Field with 64 bits of precision
```

For complex numbers, the parent depends on the PARI type:

```
sage: a = pari('(3+I)').python(); a
i + 3
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: a = pari('2^31-1').python(); a
2147483647
sage: a.parent()
```

```
Integer Ring

sage: a = pari('12/34').python(); a
6/17
sage: a.parent()
Rational Field

sage: a = pari('(3+I)/2').python(); a
1/2*i + 3/2
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: z = pari(CC(1.0+2.0*I)); z
1.00000000000000 + 2.00000000000000*I
sage: a = z.python(); a
1.00000000000000000 + 2.00000000000000000*I
sage: a.parent()
Complex Field with 64 bits of precision

sage: I = sqrt(-1)
sage: a = pari(1.0 + 2.0*I).python(); a
1.00000000000000000 + 2.00000000000000000*I
sage: a.parent()
Complex Field with 64 bits of precision
```

Vectors and matrices:

```
sage: a = pari('[1,2,3,4]')
sage: a
[1, 2, 3, 4]
sage: a.type()
't_VEC'
sage: b = a.python(); b
[1, 2, 3, 4]
sage: type(b)
<type 'list'>

sage: a = pari('[1,2;3,4]')
sage: a.type()
't_MAT'
sage: b = a.python(); b
[1 2]
[3 4]
sage: b.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

sage: a = pari('Vecsmall([1,2,3,4])')
sage: a.type()
't_VECSMALL'
sage: a.python()
[1, 2, 3, 4]
```

We use the locals dictionary:

```
sage: f = pari('(2/3)*x^3 + x - 5/7 + y')
sage: x,y=var('x,y')
sage: from sage.libs.pari.gen import gentoobj
sage: gentoobj(f, {'x':x, 'y':y})
```

```
2/3*x^3 + x + y - 5/7
sage: gentoobj(f)
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Conversion of p-adics:

```
sage: K = Qp(11,5)
sage: x = K(11^-10 + 5*11^-7 + 11^-6); x
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: y = pari(x); y
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: y.sage()
11^-10 + 5*11^-7 + 11^-6 + O(11^-5)
sage: pari(K(11^-5)).sage()
11^-5 + O(11^0)
```

Conversion of infinities:

```
sage: pari('oo').sage()
+Infinity
sage: pari('-oo').sage()
-Infinity
```

sizebyte (*x*)

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)

EXAMPLE:

```
sage: pari('1').sizebyte()
12           # 32-bit
24           # 64-bit
```

sizedigit (*x*)

sizedigit(x): Return a quick estimate for the maximal number of decimal digits before the decimal point of any component of x.

INPUT:

- x - gen

OUTPUT: Python integer

EXAMPLES:

```
sage: x = pari('10^100')
sage: x.Str().length()
101
sage: x.sizedigit()
doctest:...: DeprecationWarning: sizedigit() is deprecated in PARI
See http://trac.sagemath.org/18203 for details.
101
```

Note that digits after the decimal point are ignored:

```
sage: x = pari('1.234')
sage: x
1.23400000000000
sage: x.sizedigit()
1
```

The estimate can be one too big:

```
sage: pari('7234.1').sizedigit()
4
sage: pari('9234.1').sizedigit()
5
```

sizeword (*x*)

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:

```
sage: pari('0').sizeword()
2
sage: pari('1').sizeword()
3
sage: pari('1000000').sizeword()
3
sage: pari('10^100').sizeword()
13       # 32-bit
8        # 64-bit
sage: pari(RDF(1.0)).sizeword()
4        # 32-bit
3        # 64-bit
sage: pari('x').sizeword()
9
sage: pari('x^20').sizeword()
66
sage: pari('[x, I]').sizeword()
20
```

sqrtn (*x*, *n*, *precision=0*)

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:

```
sage: s, z = pari(2).sqrtn(5)
sage: z
0.309016994374947 + 0.951056516295154*I
sage: s
1.14869835499704
sage: s^5
2.00000000000000
sage: z^5
1.00000000000000 - 2.710505431 E-20*I       # 32-bit
1.00000000000000 - 2.71050543121376 E-20*I  # 64-bit
sage: (s*z)^5
2.00000000000000 + 0.E-19*I
```

sumdiv (*n*)

Return the sum of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdiv()
18
```

sumdivk (*n*, *k*)

Return the sum of the k-th powers of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdivk(2)
130
```

truncate (*x*, *estimate=False*)

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:

```
sage: pari('(x^2+1)/x').round()
(x^2 + 1)/x
sage: pari('(x^2+1)/x').truncate()
x
sage: pari('1.043').truncate()
1
sage: pari('1.043').truncate(True)
(1, -5)
sage: pari('1.6').truncate()
1
sage: pari('1.6').round()
2
sage: pari('1/3 + 2 + 3^2 + O(3^3)').truncate()
34/3
sage: pari('sin(x+O(x^10))').truncate()
1/362880*x^9 - 1/5040*x^7 + 1/120*x^5 - 1/6*x^3 + x
sage: pari('sin(x+O(x^10))').round()    # each coefficient has abs < 1
x + O(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:

```
sage: pari(7).type()
't_INT'
sage: pari('x').type()
't_POL'
sage: pari('oo').type()
't_INFINITY'
```

vecmax (*x*)

Return the maximum of the elements of the vector/matrix x.

EXAMPLES:

```
sage: pari([1, -5/3, 8.0]).vecmax()
8.00000000000000
```

vecmin (*x*)

Return the minimum of the elements of the vector/matrix x.

EXAMPLES:

```
sage: pari([1, -5/3, 8.0]).vecmin()
-5/3
```

class sage.libs.pari.gen.**gen_auto**

Bases: object

Part of the *gen* class containing auto-generated functions.

This class is not meant to be used directly, use the derived class `gen` instead.

Col (*x*, *n=0*)

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 $\|n\|$.

Note that the function `Colrev` does not exist, use `Vecrev` .

Colrev (*x*, *n=0*)

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 (*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. Note that, except when x is omitted, this function creates a small memory leak; so, either initialize all lists to the empty list, or use them sparingly.

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 $2x2$ matrix. Otherwise, this creates a $1x1$ 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]]~
? Mat(%)
%5 =
[1 2]

[3 4]
? Mat(Qfb(1,2,3))
%6 =
[1 1]

[1 3]
```

Mod (a, b)

In its basic form, creates 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 + O(3^6), 3)
%4 = Mod(1, 3)
? Mod(Mod(1,12), 9)
%5 = Mod(1, 3)
? Mod(1/x, x^2+1)
%6 = Mod(-1, 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
%1 = 5
? x = Mod(4,5); x + 1
%2 = 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 (*t*, *v=None*)

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 non-negative 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 $a_nX^n + ... + a_0$, 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=None*)

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=None*, *precision=0*)

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=None*, *serprec=-1*)

Transforms the object s into a power series with main variable v (x by default) and precision (number of significant terms) equal to $d >= 0$ ($d = seriesprecision$ 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

```
? Ser(1, 'y, 5)
%1 = 1 + O(y^5)
? Ser(x^2,, 5)
%2 = x^2 + O(x^7)
? T = polcyclo(100)
%3 = x^40 - x^30 + x^20 - x^10 + 1
? Ser(T, 'x, 11)
%4 = 1 - x^10 + O(x^11)
```

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

If s is a vector, on the other hand, the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (as in `Polrev` (x)), and the precision d is ignored: in other words, in this case, we convert `t_VEC` / `t_COL` to the power series whose significant terms are exactly given by the vector entries. Finally, if s is already a power series in v, we return it verbatim, ignoring d again. If d significant terms are desired in the last two cases, convert/truncate to `t_POL` first.

```
? v = [1,2,3]; Ser(v, t, 7)
%5 = 1 + 2*t + 3*t^2 + O(t^3) \\ 3 terms: 7 is ignored!
? Ser(Polrev(v,t), t, 7)
%6 = 1 + 2*t + 3*t^2 + O(t^7)
? s = 1+x+O(x^2); Ser(s, x, 7)
%7 = 1 + x + O(x^2) \\ 2 terms: 7 ignored
? Ser(truncate(s), x, 7)
%8 = 1 + x + O(x^7)
```

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 (somewhat 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*)

Converts x to a string, translating each integer into a character.

```
? Strchr(97)
%1 = "a"
? Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100])
? Strchr(%)
%3 = "hello world"
```

Vec (*x*, *n=0*)

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. `Vec` is the reciprocal function of `Pol` for a polynomial and of `Ser` for a power series,
- a matrix: returns the vector of columns comprising the matrix,
- a character string: returns the vector of individual characters,
- 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 $\|n\|$. Variant: `GEN :strong:`gtovec`(GEN x)` is also available.

`Vecrev` (*x*, *n=0*)

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=0*)

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=0*)

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=0*)

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=0*)

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

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=0*)
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.

algabsdim (*al*)
Given an algebra *al* output by `alginit` or by `algtableinit` , returns the dimension of *al* over its prime subfield ($\mathbb{Q}$ or $\mathbb{F}_p$).

```
? nf = nfinit(y^3-y+1);
? A = alginit(nf, [-1,-1]);
? algabsdim(A)
%3 = 12
```

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^2-5),[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, y^2 - 5), x^2 - 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 an 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 $2x2$ 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 (*al*, *z*, *maps=0*)
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: $\mathbb{F}_2 \oplus \mathbb{F}_4$, 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=None*)
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 , 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 , 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
```

Also accepts a square matrix with coefficients in *al*.

algdecomposition (*al*)
al being a table algebra output by algtableinit , returns $[J, [al_1, ..., al_n]]$ where J is a basis of the Jacobson radical of *al* and $al_1, ..., al_n$ are the simple factors of the semisimple algebra al/J.

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=0*)
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 non-zero 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*)

Given a central simple algebra *al* output by `alginit`, returns the dimension of *al* over its center. Given a table algebra *al* output by `algtableinit`, returns the dimension of *al* over its prime subfield ($\mathbb{Q}$ or $\mathbb{F}_p$).

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

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 $\mathrm{Tr} : \mathrm{O}_0\mathrm{x}\mathrm{O}_0 \to \mathbb{Z}$.

```
? 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 = algabsdim(A);
? 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 $x\backslash y$ 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*, return xy^{-1}. Also accepts matrices with coefficients in *al*.

alggroup (*gal*, *p=None*)
Initialize 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
```

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*.

```
? 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=None*)

Return 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 $M_d(D)$. 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=None*, *flag=1*)

Initialize 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 $K[e_1, ..., e_n]$ describes a central simple algebra over K.

```
{ 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, 0;
 0, 0,-1, 0;
 0, 1, 0, 0;
 1, 0, 0,-1];
 A = alginit(nfinit(y), [matid(4), m_i,m_j,m_k], 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^2+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^3 - 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 $flag = 0$ 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 same splitting field L in `nfinit` format.
- 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 `algord` .
- A basis of the natural order expressed on the basis of O_0, accessed with `alginvord` .
- 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 (*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=None*)

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 $e_i(e_j e_k) = (e_i e_j)e_k$ 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=None*)

Given a central simple algebra *al* output by `alginit` , test 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 test 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
```

algisramified (*al*, *pl=None*)

Given a central simple algebra *al* output by `alginit` , test 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 test 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=0*)
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 sum F_4+F_2
? algissimple(A)
%13 = 0
```

algissplit (*al*, *pl=None*)
Given a central simple algebra *al* output by `alginit`, test 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 test 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
```

alglathnf (*al*, *m*)
Given an algebra *al* and a square invertible matrix *m* with size the dimension of *al*, returns the lattice generated by the columns of *m*.

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

algleftmultable (*al*, *x*)
Given an element *x* in *al*, computes its left multiplication table. If *x* is given in basis form, returns its multiplication table on the integral basis; if *x* is given in algebraic form, returns its multiplication table on

the basis corresponding to the algebraic form of elements of *al*. In every case, if *x* is a `t_COL` of length n, then the output is a nxn `t_MAT` . Also accepts a square matrix with coefficients in *al*.

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

[1 0 1 1]

[0 0 1 1]

[0 0 -2 -1]
```

algmul (*al*, *x*, *y*)
Given two elements x and y in *al*, computes their product $x * y$ 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 =
[0 -1 1 0]

[1 0 1 1]

[0 0 1 1]

[0 0 -2 -1]
```

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*)
Given an element *x* in *al*, computes its norm. If *al* is a table algebra output by `algtableinit` , 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` , 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
```

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*.

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*, *flag=0*)
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 $flag = 1$, 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 $A = \mathbb{Q}[x]/(x^2)$, generated by $(1, x)$:

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

[1]
```

Another one with $2x2$ upper triangular matrices over $\mathbb{Q}$, 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]
```

algramifiedplaces (*al*)

Given a central simple algebra *al* output by `alginit` , return 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*, *flag=0*)

al being the output of `algtableinit` representing a semisimple algebra, returns a `t_VEC` $[al_1, al_2, ..., al_n]$ such that *al* is isomorphic to the direct sum of the simple algebras al_i. When $flag = 1$, each component is instead a `t_VEC` $[al_i, proj_i, lift_i]$ where $proj_i$ and $lift_i$ are matrices respectively representing the projection map on the i-th factor and a section of it. 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 $\mathbb{Q}$.

Warning. The images of the $lift_i$ are not guaranteed to form a direct sum.

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
```

algsplittingmatrix (*al*, *x*)

A central simple algebra *al* output by `alginit` contains data describing an isomorphism $\phi : A \otimes_K L \to M_d(L)$, where d is the degree of the algebra and L is an extension of L with $[L : K] = d$. Returns the matrix $\phi(x)$.

```
? A = alginit(nfinit(y), [-1,-1]);
? algsplittingmatrix(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)]
```

Also accepts matrices with coefficients in *al*.

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, returns an algebra isomorphic to *B*.

```
? 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)
%5 = 2
```

algtableinit (*mt*, *p=None*)

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

```
? 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=1*)

Given two algebras *al1* and *al2*, computes their tensor product. For table algebras output by `algtableinit`, the flag *maxord* is ignored. For central simple algebras output by `alginit`, 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.

algtrace (*al*, *x*)

Given an element *x* in *al*, computes its trace. If *al* is a table algebra output by `algtableinit`, returns the absolute trace of *x*, which is an element of $\mathbb{F}_p$ or $\mathbb{Q}$; if *al* is the output of `alginit`, 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
```

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;
 0,0,1,0];}
? {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
```

allocatemem (*s*)

This special operation changes the stack size *after* initialization. x must be a non-negative integer. If $x > 0$, a new stack of at least x bytes is allocated. We may allocate more than x bytes if x is way too small, or for alignment reasons: the current formula is $\max(16 * ceil x/16, 500032)$ bytes.

If $x = 0$, the size of the new stack is twice the size of the old one.

This command is much more useful if `parisizemax` is non-zero, and we describe this case first. With `parisizemax` enabled, there are three sizes of interest:

- a virtual stack size, `parisizemax`, which is an absolute upper limit for the stack size; this is set by `default(parisizemax,...)`.
- the desired typical stack size, `parisize`, that will grow as needed, up to `parisizemax`; this is set by `default(parisize,...)`.
- the current stack size, which is less that `parisizemax`, typically equal to `parisize` but possibly larger and increasing dynamically as needed; `allocatemem` allows to change that one explicitly.

The `allocatemem` command forces stack usage to increase temporarily (up to `parisizemax` of course); for instance if you notice using `\gm2` that we seem to collect garbage a lot, e.g.

```
? \gm2
 debugmem = 2
? default(parisize,"32M")
 *** Warning: new stack size = 32000000 (30.518 Mbytes).
```

```
? bnfinit('x^2+10^30-1)
 *** bnfinit: collecting garbage in hnffinal, i = 1.
 *** bnfinit: collecting garbage in hnffinal, i = 2.
 *** bnfinit: collecting garbage in hnffinal, i = 3.
```

and so on for hundred of lines. Then, provided the `breakloop` default is set, you can interrupt the computation, type `allocatemem(100*10^6)` at the break loop prompt, then let the computation go on by typing :literal:‘ <Enter> $.Backatthe : literal :$ ‘*gp* prompt, the desired stack size of `parisize` is restored. Note that changing either `parisize` or `parisizemax` at the break loop prompt would interrupt the computation, contrary to the above.

In most cases, `parisize` will increase automatically (up to `parisizemax`) and there is no need to perform the above maneuvers. But that the garbage collector is sufficiently efficient that a given computation can still run without increasing the stack size, albeit very slowly due to the frequent garbage collections.

Deprecated: when :literal:‘parisizemax. is unset‘ This is currently still the default behavior in order not to break backward compatibility. The rest of this section documents the behavior of `allocatemem` in that (deprecated) situation: it becomes a synonym for `default(parisize,...)` . In that case, there is no notion of a virtual stack, and the stack size is always equal to `parisize` . If more memory is needed, the PARI stack overflows, aborting the computation.

Thus, increasing `parisize` via `allocatemem` or `default(parisize,...)` before a big computation is important. Unfortunately, either must be typed at the `gp` prompt in interactive usage, or left by itself at the start of batch files. They cannot be used meaningfully in loop-like constructs, or as part of a larger expression sequence, e.g

```
allocatemem(); x = 1; \\ This will not set x!
```

In fact, all loops are immediately exited, user functions terminated, and the rest of the sequence following `allocatemem()` is silently discarded, as well as all pending sequences of instructions. We just go on reading the next instruction sequence from the file we are in (or from the user). In particular, we have the following possibly unexpected behavior: in

```
read("file.gp"); x = 1
```

were `file.gp` contains an `allocatemem` statement, the `x = 1` is never executed, since all pending instructions in the current sequence are discarded.

The reason for these unfortunate side-effects is that, with `parisizemax` disabled, increasing the stack size physically moves the stack, so temporary objects created during the current expression evaluation are not correct anymore. (In particular byte-compiled expressions, which are allocated on the stack.) To avoid accessing obsolete pointers to the old stack, this routine ends by a `longjmp` .

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, apply `f` on all coefficients. If `A` is a vector or list, return the elements $f(x)$ where x runs through `A` . If `A` is a matrix, return the matrix whose entries are the $f(A[i,j])$.

```
? apply(x->x^2, [1,2,3,4])
%1 = [1, 4, 9, 16]
? apply(x->x^2, [1,2;3,4])
%2 =
[1 4]

[9 16]
? apply(x->x^2, 4*x^2 + 3*x+ 2)
%3 = 16*x^2 + 9*x + 4
```

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_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=0*)
Argument of the complex number x, such that $-\pi < \arg(x) <= \pi$.

asin (*x*, *precision=0*)
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=0*)
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.

atan (*x*, *precision=0*)
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 $x! = \pm i$.

atanh (*x*, *precision=0*)
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 (*nu*, *x*, *precision=0*)
H^1-Bessel function of index *nu* and argument x.

besselh2 (*nu*, *x*, *precision=0*)
H^2-Bessel function of index *nu* and argument x.

besseli (*nu*, *x*, *precision=0*)
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=0*)
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=0*)
J-Bessel function of half integral index. More precisely, $besseljh(n,x)$ 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 **2.8.0**, this function is not very accurate when x is small.

besselk (*nu*, *x*, *precision=0*)
K-Bessel function of index *nu* and argument x.

besseln (*nu*, *x*, *precision=0*)
N-Bessel function of index *nu* and argument x.

bestappr (*x*, *B=None*)
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, return 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 $0 < v <= B$. (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 $\sqrt{N/2}$. 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 + O(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, return [].

bestapprPade (*x*, *B=-1*)
Using variants of the extended Euclidean algorithm, 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 non-negative real (impose $0 <= degree(b) <= B$).

- If x is a `t_RFRAC` or `t_SER` , this function uses continued fractions.

```
? bestapprPade((1-x^11)/(1-x)+O(x^11))
%1 = 1/(-x + 1)
? bestapprPade([1/(1+x+O(x^10)), (x^3-2)/(x^3+1)], 1)
%2 = [1/(x + 1), -2]
```

- If x is a `t_POLMOD` modulo N or a `t_SER` of precision $N = t^k$, this function performs rational modular reconstruction modulo N. The routine then returns the unique rational function a/b in coprime polynomials, with $degree(b) <= B$ 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 [].

```
? bestapprPade(Mod(1+x+x^2+x^3+x^4, x^4-2))
%1 = (2*x - 1)/(x - 1)
? % * Mod(1,x^4-2)
%2 = Mod(x^3 + x^2 + x + 3, x^4 - 2)
? bestapprPade(Mod(1+x+x^2+x^3+x^5, x^9))
%2 = []
? bestapprPade(Mod(1+x+x^2+x^3+x^5, x^10))
%3 = (2*x^4 + x^3 - x - 1)/(-x^5 + x^3 + x^2 - 1)
```

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

bezout (*x*, *y*)
Deprecated alias for `gcdext`

bezoutres (*A*, *B*, *v=None*)
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]]
```

By convention, 0 has no digits:

```
? binary(0)
%4 = []
```

binomial (*x*, *y*)

binomial coefficient $binomxy$. Here y must be an integer, but x can be any PARI object.

bitand (*x*, *y*)

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

$$\sum_i (x_i \; and \; y_i) 2^i$$

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 non-negative 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=-1*)

bitwise negation of an integer x, truncated to n bits, $n >= 0$, 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

$$\sum (x_i \; andnot(y_i)) 2^i$$

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

$$\sum (x_i \; or \; y_i) 2^i$$

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

bitprecision (*x*, *n=0*)

The function behaves differently according to whether n is present and positive 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 `+oo` .

```
? 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
```

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 + O(x), 10)
%3 = 1 + O(x)
? bitprecision(2 + O(3^5), 10)
%4 = 2 + O(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.

```
? 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=0*)

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: missing units in bnf.
? 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
```

bnfcompress (*bnf*)

Computes a compressed version of *bnf* (from `bnfinit`), a "small Buchmann's number field" (or *sbnf* for short) which contains enough information to recover a full bnf vector very rapidly, but which is much smaller and hence easy to store and print. Calling `bnfinit` on the result recovers a true `bnf` , in general different from the original. Note that an *snbf* is useless for almost all purposes besides storage, and must be converted back to *bnf* form before use; for instance, no `nf*` , `bnf*` or member function accepts them.

An *sbnf* is a 12 component vector v, as follows. Let `bnf` be the result of a full `bnfinit` , complete with units. Then $v[1]$ is `bnf.pol` , $v[2]$ is the number of real embeddings `bnf.sign[1]` , $v[3]$ is `bnf.disc` , $v[4]$ is `bnf.zk` , $v[5]$ is the list of roots `bnf.roots` , $v[7]$ is the matrix $W = bnf[1]$, $v[8]$ is the matrix $matalpha = bnf[2]$, $v[9]$ is the prime ideal factor base `bnf[5]` coded in a compact way, and ordered according to the permutation `bnf[6]` , $v[10]$ is the 2-component vector giving the number of roots of unity and a generator, expressed on the integral basis, $v[11]$ is the list of fundamental units, expressed on the integral basis, $v[12]$ is a vector containing the algebraic numbers alpha corresponding to the columns of the matrix `matalpha` , expressed on the integral basis.

All the components are exact (integral or rational), except for the roots in $v[5]$.

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[1][2]
%1 = [[Mat([8, 1]), [[0, 0, 0]]], [Mat([9, 1]), [[0, 0, 0]]]]
? bnfdecodemodule(K, s[1][1])
%2 =
[2 0]

[0 1]
```

bnfinit (*P*, *flag=0*, *tech=None*, *precision=0*)

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.

If the precision becomes insufficient, `gp` does not strive to compute the units by default ($flag = 0$).

When $flag = 1$, we insist on finding the fundamental units exactly. Be warned that this can take a very long time when the coefficients of the fundamental units on the integral basis are very large. If the fundamental units are simply too large to be represented in this form, an error message is issued. They could be obtained using the so-called compact representation of algebraic numbers as a formal product of algebraic integers. The latter is implemented internally but not publicly accessible yet.

$tech$ 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* or *sbnf* are technical and never used by the casual user. 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 $(p_i)_{1<=i<=r}$.

$bnf[2]$ 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.

$bnf[6]$ used to contain a permutation of the prime factor base, but has been obsoleted. It 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‘, a 1 (used to contain an obsolete "check number"), the number of roots of unity and a generator `:emphasis:`bnf` .tu‘, the fundamental units `:emphasis:`bnf` .fu‘.

$bnf[9]$ is a 3-element row 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 $bnf[9] = [U_r, g_a, GD_a]$.

$bnf[10]$ 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 $\mathrm{Norm}(a) = x$, 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=1*)

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, set $flag = 0$ (in this case, x is a norm iff $b = 1$). If $flag$ is non zero 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 $12\log(\mathrm{disc}(Bnf))^2$, where Bnf is the Galois closure of bnf.

See also `bnfisintnorm` .

bnfisprincipal (*bnf*, *x*, *flag=1*)

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 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 $0 <= e_i < d_i$, 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.

bnfissunit (*bnf*, *sfu*, *x*)

bnf being output by `bnfinit`, *sfu* by `bnfsunit`, gives the column vector of exponents of x on the fundamental S-units and the roots of unity. If x is not a unit, outputs an empty vector.

bnfisunit (*bnf*, *x*)

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,...,: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 $[x_1, ..., x_r, x_{r+1}]$ such that $x = u_1^{x_1}...u_r^{x_r}.\zeta^{x_{r+1}}$. The x_i are integers for $i <= r$ and is an integer modulo the order of ζ for $i = r + 1$.

Note that *bnf* need not contain the fundamental unit explicitly:

```
? setrand(1); bnf = bnfinit(x^2-x-100000);
? bnf.fu
 *** at top-level: bnf.fu
 *** ^--
 *** _.fu: missing units in .fu.
? u = [119836165644250789990462835950022871665178127611316131167, \
 379554884019013781006303254896369154068336082609238336]~;
? bnfisunit(bnf, u)
%3 = [-1, Mod(0, 2)]~
```

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

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 bnf.tufu */
tpuexpo(bnf)=
{ my(S,d,K);

 S = bnfsignunit(bnf); d = matsize(S);
 S = matrix(d[1],d[2], i,j, if (S[i,j] < 0, 1,0));
 S = concat(vectorv(d[1],i,1), S); \\ add sign(-1)
 K = lift(matker(S * Mod(1,2)));
 if (K, mathnfmodid(K, 2), 2*matid(d[1]))
}

/* totally positive units */
tpu(bnf)=
{ my(vu = bnf.tufu, ex = tpuexpo(bnf));

 vector(#ex-1, i, factorback(vu, ex[,i+1])) \\ ex[,1] is 1
}
```

bnfsunit (*bnf*, *S*, *precision=0*)

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 empty vector (used to give the logarithmic embeddings of the generators in $v[1]$ in version 2.0.16).

$v[4]$ is the S-regulator (this is the product of the regulator, the determinant of $v[2]$ and the natural logarithms of the norms of the ideals in S).

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

$v[6]$ is a copy of S.

bnrL1 (*bnr*, *H=None*, *flag=0*, *precision=0*)

Let *bnr* be the number field data output by `bnrinit(,,1)` 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,1,1);
bnrL1(bnr)
```

returns the order and the first non-zero 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,2,1);
bnrL1(bnr2,,2)
```

returns the order and the first non-zero 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=None*)

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
```

bnrclassno (*A*, *B=None*, *C=None*)

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=None*, *C=None*, *flag=0*)

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 $bnrinit(, f, 1)$.

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

bnrconductorofchar (*bnr*, *chi*)

THIS FUNCTION IS OBSOLETE: use *bnrconductor*.

bnrdisc (*A*, *B=None*, *C=None*, *flag=0*)

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 $L/\mathbb{Q}$: N is the absolute degree $[L : \mathbb{Q}]$, r_1 the number of real places of L, and D the discriminant of $L/\mathbb{Q}$. Otherwise, output relative data for L/K: N is the relative degree $[L : K]$, r_1 is the number of real places of K unramified in L (so that the number of real places of L is equal to r_1 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=None*)

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 of vectors, where $v[i][j]$ is understood to be in fact $V[2^{15}(i-1)+j]$ of a unique big vector V. (This awkward scheme allows for larger vectors than could be otherwise represented.)

$V[k]$ 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` is 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`).

See `bnrisgalois` for an example.

bnrinit (*bnf*, *f*, *flag=0*)

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`.)

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 computed, which saves time. Hence `:emphasis:`bnr` .gen‘ would produce an error.

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

bnrisconductor (*A*, *B=None*, *C=None*)

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);
G=galoisinit(K);
B=bnrinit(K,9,1);
L1=[H|H<-subgrouplist(B,3), bnrisgalois(B,G,H)]
##
M=bnrgaloismatrix(B,G)
L2=[H|H<-subgrouplist(B,3), bnrisgalois(B,M,H)]
##
```

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

bnrisprincipal (*bnr*, *x*, *flag=1*)

bnr being the number field data which is output by `bnrinit` $(,,1)$ and x being an ideal in any form, outputs the components of x on the ray class group generators in a way similar to `bnfisprincipal` . That is a 2-component vector v where $v[1]$ is the vector of components of x on the ray class group generators, $v[2]$ gives on the integral basis an element α such that $x = \alpha \prod_i g_i^{x_i}$.

If $flag = 0$, outputs only v_1. In that case, *bnr* need not contain the ray class group generators, i.e. it may be created with `bnrinit` $(,,0)$ If x is not coprime to the modulus of *bnr* the result is undefined.

bnrrootnumber (*bnr*, *chi*, *flag=0*, *precision=0*)

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.

The generators of the ray class group are needed, and 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]], 1);
bnrrootnumber(bnr, [2,1])
```

returns the root number of the character χ of $\mathrm{Cl}_{7\infty_1\infty_2}(\mathbb{Q}(\sqrt{229}))$ 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=None*, *precision=0*)

bnr being as output by `bnrinit(,,1)`, 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, 1);
bnrstark(bnr)
```

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

```
rnfpolredabs(bnf, pol, 16) \\ compute a reduced relative polynomial
rnfpolredabs(bnf, pol, 16 + 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. It was decided that it was more useful to keep the extra information thus made available, hence the user has to take the compositum herself.

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 `rnfkummer`:

```
? bnr = bnrinit(bnfinit(y^8-12*y^6+36*y^4-36*y^2+9,1), 2, 1);
? bnrstark(bnr)
 *** at top-level: bnrstark(bnr)
 *** ^-------------
 *** bnrstark: need 3919350809720744 coefficients in initzeta.
 *** Computation impossible.
? lift( rnfkummer(bnr) )
time = 24 ms.
%2 = x^2 + (1/3*y^6 - 11/3*y^4 + 8*y^2 - 5)
```

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`(:math:`f,A`)` = `apply`(:math:`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=None*)

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 non-negative; 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 $\sum_{j<=k} \mathbb{Z}/d_j\mathbb{Z}$ 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 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 `idealstar(,q)`), characters in Conrey representation are available, see `dirichletchar` (in the PARI manual) or `??character`:

```
? G = idealstar(,8); \\ (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 $a/b = a\bar{b}$.

```
? 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 = idealstar(,100);
? 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=None*)

Let G be an abelian group structure affording a discrete logarithm method, e.g $G = idealstar(, N)$ 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

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

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 `bid = idealstar(,q)`), 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 $0 <= c(x) < o$. 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.

charker (*cyc*, *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_j 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 `idealstar(,q)`), characters in Conrey representation are available, see `dirichletchar` (in the PARI manual) or `??character` .

```
? G = idealstar(,8); \\ (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 = idealstar(,100);
? 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 $\sum_{j<=k} \mathbb{Z}/d_j\mathbb{Z}$ 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 `idealstar(,q)`), characters in Conrey representation are available, see `dirichletchar` (in the PARI manual) or `??character`:

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

charpoly (*A*, *v=None*, *flag=5*)

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 $flag = 2$).

chinese (*x*, *y=None*)

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.

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 $x\ ===\ y$, and its restriction to integers coincides with the customary one. Besides that, it has no useful mathematical meaning.

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 + O(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=None*)

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 `[ ]~`.)

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 (*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=0*)

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*)

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.

contfrac (*x*, *b=None*, *nmax=0*)

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 $\deg a_i = \deg b_i + 1$. 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=-1*)

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=-1*)

Given M representing the power series $S = \sum_{n>=0} M[n+1]z^n$, transform it into a continued fraction; restrict to $n <= lim$ if latter is non-negative. M can be a vector, a power series, a polynomial, or a rational function. 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=-1*)

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 $n >= 0$ 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,20)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2]
? contfracpnqn(a,3)
%2 =
[3 22 333 355]

[1 7 106 113]

? contfracpnqn(a,7)
%3 =
[3 22 333 355 103993 104348 208341 312689]

[1 7 106 113 33102 33215 66317 99532]
```

core (*n*, *flag=0*)

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

coredisc (*n*, *flag=0*)

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 non-zero integer n, this routine returns the (unique) fundamental discriminant d such that $n = df^2$, f a positive rational number. If $flag$ is non-zero, 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=0*)
Cosine of x.

cosh (*x*, *precision=0*)
Hyperbolic cosine of x.

cotan (*x*, *precision=0*)
Cotangent of x.

cotanh (*x*, *precision=0*)
Hyperbolic cotangent of x.

denominator (*x*)
Denominator of x. The meaning of this is clear when x is a rational number or function. If x 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(x) )
```

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 (minimal degree) polynomial representative.

If x 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).

deriv (*x*, *v=None*)
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.

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.

diffop (*x*, *v*, *d*, *n=1*)
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.

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

```
? 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 follow, 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(var(i)=eval(Str("X",i)));
 my(x,v,dv);
 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),polcoeff(x,n,'X))
}
```

digits (*x*, *b=None*)

Outputs the vector of the digits of $\|x\|$ in base b, where x and b are integers ($b = 10$ by default). See `fromdigits` for the reverse operation.

```
? digits(123)
%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=0*)

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 non-zero coefficient.

```
? dirmul([0,1], [0,1]);
%2 = [0, 0, 0, 1]
```

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*)

Creates a row vector whose components are the divisors of x. The factorization of x (as output by `factor`) can be used instead.

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.

divrem (*x*, *y*, *v=None*)

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:

```
? divrem(1/2, 3)[2]
%1 = 1/2
? (1/2) % 3
%2 = 2
? divrem(Mod(2,9), 3)[2]
 *** at top-level: divrem(Mod(2,9),3)[2
 *** ^--------------------
 *** forbidden division t_INTMOD \ t_INT.
? Mod(2,9) % 6
%3 = Mod(2,3)
```

eint1 (*x*, *n=None*, *precision=0*)

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.

ellL1 (*e*, *r=0*, *precision=0*)

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 = 10.391099400715804138751850510360917069726356375657009279 7[...]
```

elladd (*E*, *z1*, *z2*)
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 `ellinit` . E must be given by an integral model, not necessarily minimal, although a minimal model will make the function faster.

```
? E = ellinit([0,1]);
? ellak(E, 10)
%2 = 0
? e = ellinit([5^4,5^6]); \\ not minimal at 5
? ellak(e, 5) \\ 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 = 820 ms.
? for(i=1,10^6, ellak(e,5)) \\ 5 is bad, markedly slower
time = 1,249 ms.

? for(i=1,10^5,ellak(E,5*i))
time = 977 ms.
? for(i=1,10^5,ellak(e,5*i)) \\ still slower but not so much on average
time = 1,008 ms.
```

ellanalyticrank (*e*, *eps=None*, *precision=0*)
Returns the order of vanishing at $s = 1$ of the L-function of the elliptic curve e and the value of the first non-zero derivative. To determine this order, it is assumed that any value less than `eps` is zero. If no value of `eps` is given, a value of half the current precision is used.

```
? 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=None*)
Let E be an `ell` structure as output by `ellinit` , defined over a number field or a finite field $\mathbb{F}_q$. The argument p is best left omitted if the curve is defined over a finite field, and must be a prime number or a maximal ideal otherwise. This function computes the trace of Frobenius t for the elliptic curve E, defined

by the equation $\#E(\mathbb{F}_q) = q + 1 - t$ (for primes of good reduction).

When the characteristic of the finite field is large, the availability of the `seadata` package will speed the computation.

If the curve is defined over $\mathbb{Q}$, p must be explicitly given and the function computes the trace of the reduction over $\mathbb{F}_p$. The trace of Frobenius is also the a_p coefficient in the curve L-series $L(E, s) = \sum_n a_n n^{-s}$, whence the function name. The equation must be integral at p but need not be minimal at p; of course, a minimal model will be more efficient.

```
? E = ellinit([0,1]); \\ y^2 = x^3 + 0.x + 1, defined over Q
? ellap(E, 7) \\ 7 necessary here
%2 = -4 \\ #E(F_7) = 7+1-(-4) = 12
? ellcard(E, 7)
%3 = 12 \\ OK

? E = ellinit([0,1], 11); \\ defined over F_11
? ellap(E) \\ no need to repeat 11
%4 = 0
? ellap(E, 11) \\ ... 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

? Fq = ffgen(ffinit(11,3), 'a); \\ defines F_q := F_{11^3}
? E = ellinit([a+1,a], Fq); \\ 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. If p is above 2 or 3, the function currently assumes (without checking) that the given model is locally minimal at p. There is no restriction at other primes.

```
? 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 \\ non-split multiplicative reduction
? ellap(E, fa[2,1])
%4 = 1 \\ split multiplicative reduction
? P17 = idealprimedec(K,17)[1];
? ellap(E, P17)
%6 = 6 \\ good reduction
? E2 = ellchangecurve(E, [17,0,0,0]);
? ellap(E2, P17)
%8 = 6 \\ same, starting from a non-miminal model

? P3 = idealprimedec(K,3)[1];
? E3 = ellchangecurve(E, [3,0,0,0]);
? ellap(E, P3) \\ OK: E is minimal at P3
%11 = -2
? ellap(E3, P3) \\ junk: E3 is not minimal at P3 | 3
```

```
%12 = 0
```

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 $O(\log q)^2$. 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=0*)
Deprecated alias for `ellheight(E,P,Q)` .

ellcard (*E*, *p=None*)
Let E be an `ell` structure as output by `ellinit` , defined over $\mathbb{Q}$ or a finite field $\mathbb{F}_q$. The argument p is best left omitted if the curve is defined over a finite field, and must be a prime number otherwise. This function computes the order of the group $E(\mathbb{F}_q)$ (as would be computed by `ellgroup`).

When the characteristic of the finite field is large, the availability of the `seadata` package will speed the computation.

If the curve is defined over $\mathbb{Q}$, p must be explicitly given and the function computes the cardinality of the reduction over $\mathbb{F}_p$; the equation need not be minimal at p, but a minimal model will be more efficient. The reduction is allowed to be singular, and we return the order of the group of non-singular points in this case.

ellchangecurve (*E*, *v*)
Changes the data for the elliptic curve E by changing the coordinates using the vector `v = [u,r,s,t]` , i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. 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 `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 (*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=None*)

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, 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

$$f_1 = 1, f_2 = 4X^3 + b_2X^2 + 2b_4X + b_6, f_3 = 3X^4 + b_2X^3 + 3b_4X^2 + 3b_6X + b8,$$

$$f_4 = f_2(2X^6 + b_2X^5 + 5b_4X^4 + 10b_6X^3 + 10b_8X^2 + (b_2b_8 - b_4b_6)X + (b_8b_4 - b_6^2)), ...$$

For $n >= 2$, the roots of f_n are the X-coordinates of points in $E[n]$.

elleisnum (*w*, *k*, *flag=0*, *precision=0*)

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`.

```
? w = ellperiods([1,I]);
? elleisnum(w, 4)
%2 = 2268.8726415508062275167367584190557607
? elleisnum(w, 6)
%3 = -3.977978632282564763 E-33
? E = ellinit([1, 0]);
? elleisnum(E, 4, 1)
%5 = -47.999999999999999999999999999999999998
```

When *flag* is non-zero 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=0*)

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 = [3.1415926535897932384626433832, 9.4247779607693797153879301498*I]
```

ellformaldifferential (*E*, *serprec=-1*, *n=None*)

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 $\omega = f(t)dt, \eta = g(t)dt$:

$$f(t) = 1 + a_1t + (a_1^2 + a_2)t^2 + ..., g(t) = t^{-2} + ...$$

```
? E = ellinit([-1,1/4]); [f,g] = ellformaldifferential(E,7,'t);
? f
%2 = 1 - 2*t^4 + 3/4*t^6 + O(t^7)
? g
%3 = t^-2 - t^2 + 1/2*t^4 + O(t^5)
```

ellformalexp (*E*, *serprec=-1*, *n=None*)

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`): $Exp o L = \mathrm{Id}$. 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 + O(z^11)
? L = ellformallog(E,10,'t);
? subst(Exp,z,L)
%3 = t + O(t^11)
```

ellformallog (*E*, *serprec=-1*, *n=None*)

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 + O(t^10)
? [f,g] = ellformaldifferential(E,8,'t);
? L' - f
%3 = O(t^8)
```

ellformalpoint (*E*, *serprec=-1*, *n=None*)

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*:

$$x = t^{-2} - a_1t^{-1} - a_2 - a_3t + ...$$

$$y = -t^{-3} - a_1t^{-2} - a_2t^{-1} - a_3 + ...$$

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 + O(t^6)
? y
%3 = -t^-3 + 1 - t + t^3 - 2*t^4 + O(t^5)
? E = ellinit([0,1/2]); ellformalpoint(E,7)
%4 = [x^-2 - 1/2*x^4 + O(x^5), -x^-3 + 1/2*x^3 + O(x^4)]
```

ellformalw (*E*, *serprec=-1*, *n=None*)

Return the formal power series w attached to the elliptic curve E, in the variable t:

$$w(t) = t^3 + a_1t^4 + (a_2 + a_1^2)t^5 + ... + O(t^{n+3}),$$

which is the formal expansion of $-1/y$ in the formal parameter $t := -x/y$ at oo (take $n = seriesprecision$ 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 + O(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(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 (*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 $E(\mathbb{F}_q)$.

ellglobalred (*E*)

Calculates the arithmetic conductor, the global minimal model of E and the global Tamagawa number c. E must be an `ell` structure as output by `ellinit` , defined over $\mathbb{Q}$. The result is a vector $[N, v, c, F, L]$, where

- N is the arithmetic conductor of the curve,
- v gives the coordinate change for E over $\mathbb{Q}$ to the minimal integral model (see `ellminimalmodel`),
- 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 over $\mathbb{Z}$.
- L is a vector, whose i-th entry contains the local data at the i-th prime divisor of N, i.e. `L[i] = elllocalred(E,F[i,1])`, where the local coordinate change has been deleted and replaced by a 0.

ellgroup (*E*, *p=None*, *flag=0*)

Let E be an `ell` structure as output by `ellinit`, defined over $\mathbb{Q}$ or a finite field $\mathbb{F}_q$. The argument p is best left omitted if the curve is defined over a finite field, and must be a prime number otherwise. This function computes the structure of the group $E(\mathbb{F}_q)$ $\mathbb{Z}/d_1\mathbb{Z}x\mathbb{Z}/d_2\mathbb{Z}$, with $d_2\|d_1$.

If the curve is defined over $\mathbb{Q}$, p must be explicitly given and the function computes the structure of the reduction over $\mathbb{F}_p$; the equation need not be minimal at p, but a minimal model will be more efficient. The reduction is allowed to be singular, and we return the structure of the (cyclic) group of non-singular points in this case.

If the flag is 0 (default), return $[d_1]$ or $[d_1, d_2]$, if $d_2 > 1$. If the flag is 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,d1)` has order d_2.

```
? 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, non-cyclic
? 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)]]]
```

If E is defined over $\mathbb{Q}$, we allow singular reduction and in this case we return the structure of the group of non-singular 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 \\ non-split 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 non-torsion 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).

```
? E = ellinit([-157^2,0]);
? u = ellheegner(E); print(u[1], "\n", u[2])
69648970982596494254458225/166136231668185267540804
538962435089604615078004307258785218335/67716816556077455999228495435742408
? ellheegner(ellinit([0,1])) \\ E has rank 0 !
 *** at top-level: ellheegner(E=ellinit
 *** ^--------------------
 *** ellheegner: The curve has even analytic rank.
```

ellheight (*E*, *P*, *Q=None*, *precision=0*)

Global Néron-Tate height $h(P)$ of the point P on the elliptic curve $E/\mathbb{Q}$, using the normalization in Cremona's *Algorithms for modular elliptic curves*. E must be an `ell` as output by `ellinit` ; it needs not be given by a minimal model although the computation will be faster if it is.

If the argument Q is present, computes the value of the bilinear form $(h(P+Q)-h(P-Q))/4$.

ellheightmatrix (*E*, *x*, *precision=0*)

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 change, suitable for `ellchangecurve` .

ellinit (*x*, *D=None*, *precision=0*)

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

$$Y^2+a_1XY+a_3Y=X^3+a_2X^2+a_4X+a_6,$$

- a 2-component vector $[a_4,a_6]$ defining the elliptic curve with short Weierstrass equation

$$Y^2=X^3+a_4X+a_6,$$

- 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 $\mathbb{Q}$.
- 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`, $O(p^n)$: the field $\mathbb{Q}_p$, where p-adic quantities will be computed to a relative accuracy of n digits. We advise to input a model defined over $\mathbb{Q}$ for such curves. In any case, if you input an approximate model with `t_PADIC` coefficients, it will be replaced by a lift to $\mathbb{Q}$ (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:

```
? ellinit([1 + O(5), 1], O(7));
 *** at top-level: ellinit([1+O(5),1],O
 *** ^--------------------
 *** ellinit: inconsistent moduli in ellinit: 7 != 5
```

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.

ellisogeny (*E*, *G*, *only_image=0*, *x=None*, *y=None*)

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 non-zero 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/G and (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*, *fl=0*)

Given an elliptic curve E defined over $\mathbb{Q}$, compute representatives of the isomorphism classes of elliptic curves $\mathbb{Q}$-isogenous to E. The function returns a vector $[L, M]$ where L is a list of triples $[E_i, f_i, g_i]$, where E_i is an elliptic curve in $[a_4, a_6]$ form, $f_i : E \to E_i$ is a rational isogeny, $g_i : E_i \to E$ is the dual isogeny of f_i, and M is the matrix such that $M_{i,j}$ is the degree of the isogeny between E_i and E_j. Furthermore the first curve E_1 is isomorphic to E by f_1. If the flag $fl = 1$, the f_i and g_i are not computed, which saves time, and L is the list of the curves E_i.

```
? E = ellinit("14a1");
? [L,M] = ellisomat(E);
? LE = apply(x->x[1], L) \\ 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] \\ 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.

ellissupersingular (*E*, *p=None*)

Return 1 if the elliptic curve E defined over Q or a finite field is supersingular at p, and 0 otherwise. If the curve is defined over $\mathbb{Q}$, p must be explicitly given and have good reduction at p. Alternatively, E can be given byt its j-invariant in a finite field. In this case p must be omitted.

```
? g = ffprimroot(ffgen(7^5))
%1 = x^3 + 2*x^2 + 3*x + 1
? [g^n | n <- [1 .. 7^5 - 1], ellissupersingular(g^n)]
%2 = [6]
```

ellj (*x*, *precision=0*)

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 $\mathbb{Q}$ (p a rational prime) or a number field K (p a maximal ideal given by a `prid` structure), and is assumed to have all its coefficients a_i integral. 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=None*)

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 non-negative 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=None*, *precision=0*)

E being an elliptic curve, given by an arbitrary model over $\mathbb{Q}$ as output by `ellinit` , 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.

ellminimaltwist (*E*, *flag=0*)

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
```

ellmoddegree (*e*, *precision=0*)

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. Return $[D, err]$, where D is a rational number and err is exponent of the truncation error.

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).

```
? 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)
 *** ^--------------
 *** 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 (*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 non-singular.

```
? 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=None*)
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]); \\ 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 ms.
```

The parameter o, is now mostly useless, and kept for backward compatibility. If present, it represents a non-zero 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=0*)

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=None*, *r=0*, *D=None*)

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 $< \chi >= \chi\tau^{-1}$, with values in $1 + 2p\mathbb{Z}_p$. In GP, the continuous character of $Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$ given by $< \chi >^{s_1} \tau^{s_2}$ is represented by the pair of integers $s = (s_1, s_2)$, with $s_1 \in \mathbb{Z}_p$ and $s_2 mod p - 1$ 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 $\mathbb{Q}_p \otimes H^1_{dR}(E/\mathbb{Q})$ and satisfies

$$where : math : `F`istheFrobenius, : math : `L(E,1)`isthevalueofthecomplex : math : `L`functionat : math : `1`, :$$

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 + O(3^5), O(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 + O(3^4), O(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 + O(5^4)
? al = padicappr(x^2 - ap*x + p, ap + O(p^7))[1];
? (1-al^(-1))^(-2) * L / cxL(e)
%4 = 1 + O(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)
```

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 + O(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 + O(5^6)
? mspadicseries(mu)
%7 = (4 + 3*5 + 4*5^2 + 2*5^3 + 2*5^4 + 5^5 + O(5^6))
 + (3 + 3*5 + 5^2 + 5^3 + O(5^4))*x
 + (2 + 3*5 + 5^2 + O(5^3))*x^2
 + (3 + 4*5 + 4*5^2 + O(5^3))*x^3
 + (3 + 2*5 + O(5^2))*x^4 + O(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.

ellpadicfrobenius (*E, p, n*)

If $p > 2$ is a prime and E is a 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 $x^2 - a_px + p$. 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 + O(5^3)*x + (5 + O(5^3))
? ellap(E, 5)
%5 = 0
```

ellpadicheight (*E, p, n, P, Q=None*)

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 $D_{dR}(E) := H^1_{dR}(E) \otimes_{\mathbb{Q}} \mathbb{Q}_p$ be the $\mathbb{Q}_p$ vector space spanned by ω (invariant differential $dx/(2y + a_1x + a3)$ related to the given model) and $\eta = x\omega$. Then the cyclotomic p-adic height associates to $P \in E(\mathbb{Q})$ an element $f\omega + g\eta$ in D_{dR}. 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,5,4, P)
%2 = [3*5 + 5^2 + 2*5^3 + O(5^4), 5^2 + 4*5^4 + O(5^6)]
? E = ellinit("11a1"); P = [5,5]; \\ torsion point
? ellpadicheight(E,19,6, P)
%4 = O(19^6)
? E = ellinit([0,0,1,-4,2]); P = [-2,1];
? ellpadicheight(E,3,5, P)
%6 = [2*3^2 + 2*3^3 + 3^4 + O(3^5), 2*3^2 + 3^4 + 2*3^5 + 3^6 + O(3^7)]
? ellpadicheight(E,3,5, P, elladd(E,P,P))
```

One can replace the parameter p prime by a vector $[p, [a, b]]$, in which case the routine returns the p-adic number $af + bg$.

When E has good ordinary reduction at p, the "canonical" p-adic height is given by

```
s2 = ellpadics2(E,p,n);
ellpadicheight(E, [p,[1,-s2]], n, P)
```

Since s_2 does not depend on P, it is preferable to compute it only once:

```
? E = ellinit("5077a1"); p = 5; n = 7;
? s2 = ellpadics2(E,p,n);
? M = ellpadicheightmatrix(E,[p,[1,-s2]], n, E.gen);
? matdet(M) \\ p-adic regulator
%4 = 5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 5^6 + O(5^7)
```

ellpadicheightmatrix (*E*, *p*, *n*, *v*)

v being a vector of points, this function outputs the Gram matrix of v with respect to the cyclotomic p-adic height, given to n p-adic digits; in other words, the (i, j) component of the matrix is equal to `ellpadicheight` $(E, p, n, v[i], v[j]) = [f, g]$.

See `ellpadicheight` ; in particular one can replace the parameter p prime by a vector $[p, [a, b]]$, in which case the routine returns the matrix containing the p-adic numbers $af + bg$.

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: $dL = dx/(2y + a_1x + a_3)$.

ellpadics2 (*E*, *p*, *n*)

If $p > 2$ is a prime and $E/\mathbb{Q}$ is a 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.

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=0*, *precision=0*)

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 $\eta_1 W_2 - \eta_2 W_1 = 2i\pi$.

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.

ellpointtoz (*E*, *P*, *precision=0*)

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

$$y^2 = 4x^3 - g_2 x - g_3,$$

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 $0 <= \Im(t) < \Im(w2)$, where $(w1, w2)$ are the real and complex periods of E.

```
? E = ellinit([0,1]); P = [2,3];
? z = ellpointtoz(E, P)
%2 = 3.5054552633136356529375476976257353387
? ellwp(E, z)
%3 = 2.0000000000000000000000000000000000000
? ellztopoint(E, z) - P
%4 = [6.372367644529809109 E-58, 7.646841173435770930 E-57]
? ellpointtoz(E, [0]) \\ the point at infinity
%5 = 0
```

If $E/\mathbb{Q}_p$ has multiplicative reduction, then $E/\bar{\mathbb{Q}_p}$ is analytically isomorphic to $\bar{\mathbb{Q}}_p^*/q^{\mathbb{Z}}$ (Tate curve) for some p-adic integer q. The behaviour is then as follows:

- If the reduction is split ($E.tate[2]$ 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], 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 + O(11^5)
? z^5
%5 = 1 + O(11^5)
? 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` .

ellrootno (*E*, *p=None*)
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. 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=0*)
Let E be an *ell* structure as output by `ellinit` , defined over a finite field $\mathbb{F}_q$. This function computes the order of the group $E(\mathbb{F}_q)$ using the SEA algorithm and the `tors` argument allows to speed up a search for curves having almost prime order.

- If the characteristic is too small ($p <= 7$) the generic algorithm `ellcard` is used instead and the `tors` argument is ignored.
- When `tors` is set to a non-zero 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 $\#E(\mathbb{F}_q)$.

In particular, you should set `tors` to 1 if you want a curve with prime order, to 2 if you want to allow a cofacteur which is a power of two (e.g. for Edwards's curves), etc.

The availability of the `seadata` package will speed up the computation, and is strongly recommended.

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 `ellsea(E,1)` instead of `ellsea(E)`) runs for about 5 minutes before finding a suitable curve.

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=None*, *flag=0*, *precision=0*)

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 `ellinit` ($L = E.omega$).

```
? 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 (*E*, *z1*, *z2*)

Difference of the points $z1$ and $z2$ on the elliptic curve corresponding to E.

elltaniyama (*E*, *serprec=-1*)

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 d 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 $du/(2v + a_1 u + a_3)$ 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, $f(x) = \sum_{n>0} ellan(E,n)x^n$.

elltatepairing (*E*, *P*, *Q*, *m*)
Computes the Tate pairing of the two points P and Q on the elliptic curve E. The point P must be of m-torsion.

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=None*)
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 non-trivial twist is returned.

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-Shreier 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]
```

ellweilpairing (*E*, *P*, *Q*, *m*)
Computes the Weil pairing of the two points of m-torsion P and Q on the elliptic curve E.

ellwp (*w*, *z=None*, *flag=0*, *precision=0*)
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 `ellinit` ($w = E.omega$).

```
? 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 + O(x^14)
? ellwp(E, x + O(x^12)) \\ explicit precision
%6 = x^-2 - 1/5*x^2 + 1/75*x^6 + O(x^9)
```

Optional *flag* means 0 (default): compute only $\wp(z)$, 1: compute $[\wp(z), \wp'(z)]$.

ellxn (*E*, *n*, *v=None*)

In standard notation, for any affine point $P = (v, w)$ on the curve E, 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 $\mathbb{Z}[a_1, a_2, a_3, a_4, a_6][v, w]$. This function returns $[\phi_n(P), \psi_n(P)^2]$, which give the numerator and denominator of the abcissa of $[n]P$ and depend only on v.

ellzeta (*w*, *z=None*, *precision=0*)

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 `ellinit` ($w = E.omega$). 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 + O(x^15)
? ellzeta(E, x + O(x^20)) \\ explicit precision
%5 = x^-1 + 1/35*x^5 - 1/7007*x^11 + 1/1440257*x^17 + O(x^18)
```

ellztopoint (*E*, *z*, *precision=0*)

E being an *ell* as output by `ellinit` , computes the coordinates $[x, y]$ on the curve E corresponding to the complex number z. Hence this is the inverse function of `ellpointtoz` . In other words, if the curve is put in Weierstrass form $y^2 = 4x^3 - g_2x - g_3$, $[x, y]$ represents the Weierstrass $\wp$-function and its derivative. More precisely, we have

$$x = \wp(z) - b_2/12, y = \wp'(z) - (a_1x + a_3)/2.$$

If z is in the lattice defining E over $\mathbb{C}$, the result is the point at infinity $[0]$.

erfc (*x*, *precision=0*)

Complementary error function, analytic continuation of $(2/\sqrt{\pi})\int_x^o oe^{-t^2}dt = incgam(1/2, x^2)/\sqrt{\pi}$, where the latter expression extends the function definition from real x to all complex $x! = 0$.

errname (*E*)

Returns the type of the error message `E` as a string.

eta (*z*, *flag=0*, *precision=0*)
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 non-zero, 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 (*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 $n \in \mathbb{Z}$.

exp (*x*, *precision=0*)
Exponential of x. p-adic arguments with positive valuation are accepted.

expm1 (*x*, *precision=0*)
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 = 28 ms.
? default(realprecision, 10040); x = 1e-100;
? c = expm1(x); \\ reference point
? abs(a-c)/c \\ relative error in expm1(x)
%7 = 0.E-10017
? 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 both faster and more accurate than `exp(x)-1` .

factor (*x*, *lim=None*)
General factorization function, where x is a rational (including integers), a complex number with rational real and imaginary parts, or a rational function (including polynomials). 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:mathbb{Q}' and $\mathbb{Q}(i)$.' See `factorint` for more information about the algorithms used. The rational or Gaussian primes are in fact *pseudoprimes* (see `ispseudoprime`), a priori not rigorously proven primes. In fact, any factor which is $<= 2^{64}$ (whose norm is $<= 2^{64}$ for an irrational Gaussian prime) is a genuine prime. Use `isprime` to prove primality of other factors, as in

```
? fa = factor(2^2^7 + 1)
%1 =
[59649589127497217 1]
```

```
[5704689200685129054721 1]

? isprime( fa[,1] )
%2 = [1, 1]~ \\ both entries are proven primes
```

Another possibility is to set the global default `factor_proven` , which will perform a rigorous primality proof for each pseudoprime factor.

A `t_INT` argument *lim* can be added, meaning that we look only for prime factors $p < lim$. The limit *lim* must be non-negative. In this case, all but the last factor are proven primes, but the remaining factor may actually be a proven composite! If the remaining factor is less than lim^2, then it is prime.

```
? factor(2^2^7 +1, 10^5)
%3 =
[340282366920938463463374607431768211457 1]
```

Deprecated feature. Setting $lim = 0$ is the same as setting it to $primelimit + 1$. Don't use this: it is unwise to rely on global variables when you can specify an explicit argument.

This routine uses trial division and perfect power tests, and should not be used for huge values of *lim* (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 much more efficient way.

```
? F = (2^2^7 + 1) * 1009 * 100003; factor(F, 10^5) \\ fast, incomplete
time = 0 ms.
%4 =
[1009 1]

[34029257539194609161727850866999116450334371 1]

? factor(F, 10^9) \\ very slow
time = 6,892 ms.
%6 =
[1009 1]

[100003 1]

[340282366920938463463374607431768211457 1]

? factorint(F, 1+8) \\ much faster, all small primes were found
time = 12 ms.
%7 =
[1009 1]

[100003 1]

[340282366920938463463374607431768211457 1]

? factor(F) \\ complete factorisation
time = 112 ms.
%8 =
[1009 1]

[100003 1]

[59649589127497217 1]
```

```
[5704689200685129054721 1]
```

Over $\mathbb{Q}$, the prime factors are sorted in increasing order.

Rational functions. The polynomials or rational functions to be factored must have scalar coefficients. In particular PARI does not know how to factor *multivariate* polynomials. The following domains 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, `factornf` for the algorithms over number fields. Over $\mathbb{Q}$, van Hoeij's method is used, which is able to cope with hundreds of modular factors.

The routine guesses a sensible ring over which to factor: the smallest ring 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). Factoring modulo a non-prime number is not supported; to factor in $\mathbb{Q}_p$, use `t_PADIC` coefficients not `t_INTMOD` modulo p^n.

```
? 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 + O(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})
```

In most cases, it is clearer and simpler to call an explicit variant than to rely on the generic `factor` function and the above detection mechanism:

```
? factormod(T, 3) \\ over F_3
? factorff(T, 3, t^2+t+2)) \\ 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
```

Note that factorization of polynomials is done up to multiplication by a constant. In particular, the factors of rational polynomials will have integer coefficients, and the content of a polynomial or rational function is discarded and not included in the factorization. If needed, you can always ask for the content explicitly:

```
? factor(t^2 + 5/2*t + 1)
%1 =
[2*t + 1 1]

[t + 2 1]

? content(t^2 + 5/2*t + 1)
%2 = 1/2
```

The irreducible factors are sorted by increasing degree. See also `nffactor` .

factorback (*f*, *e=None*)

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*)

Factors the polynomial x modulo the prime p, using distinct degree plus Cantor-Zassenhaus. The coefficients of x must be operation-compatible with $\mathbb{Z}/p\mathbb{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If you want only the *degrees* of the irreducible polynomials (for example for computing an L-function), use $factormod(x, p, 1)$. Note that the `factormod` algorithm is usually faster than `factorcantor`.

factorff (*x*, *p=None*, *a=None*)

Factors the polynomial x in the field $\mathbb{F}_q$ defined by the irreducible polynomial a over $\mathbb{F}_p$. The coefficients of x must be operation-compatible with $\mathbb{Z}/p\mathbb{Z}$. The result is a two-column matrix: the first column contains the irreducible factors of x, and the second their exponents. If all the coefficients of x are in $\mathbb{F}_p$, a much faster algorithm is applied, using the computation of isomorphisms between finite fields.

Either a or p can omitted (in which case both are ignored) if x has `t_FFELT` coefficients; the function then becomes identical to `factor`:

```
? factorff(x^2 + 1, 5, y^2+3) \\ over F_5[y]/(y^2+3) ~ F_25
%1 =
[Mod(Mod(1, 5), Mod(1, 5)*y^2 + Mod(3, 5))*x
 + Mod(Mod(2, 5), Mod(1, 5)*y^2 + Mod(3, 5)) 1]

[Mod(Mod(1, 5), Mod(1, 5)*y^2 + Mod(3, 5))*x
 + Mod(Mod(3, 5), Mod(1, 5)*y^2 + Mod(3, 5)) 1]
? t = ffgen(y^2 + Mod(3,5), 't); \\ a generator for F_25 as a t_FFELT
? factorff(x^2 + 1) \\ not enough information to determine the base field
 *** at top-level: factorff(x^2+1)
 *** ^---------------
 *** factorff: incorrect type in factorff.
? factorff(x^2 + t^0) \\ make sure a coeff. is a t_FFELT
%3 =
[x + 2 1]

[x + 3 1]
? factorff(x^2 + t + 1)
%11 =
[x + (2*t + 1) 1]

[x + (3*t + 4) 1]
```

Notice that the second syntax is easier to use and much more readable.

factorint (*x*, *flag=0*)

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 is 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 (*x*, *p*, *flag=0*)

Factors the polynomial x modulo the prime integer p, using Berlekamp. The coefficients of x must be operation-compatible with $\mathbb{Z}/p\mathbb{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If $flag$ is non-zero, outputs only the *degrees* of the irreducible polynomials (for example, for computing an L-function). A different algorithm for computing the mod p factorization is `factorcantor` which is sometimes faster.

factornf (*x*, *t*)

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 non-rational 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 (*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 + O(3^5))*x^2 + O(3^5)*x + (3^2 + O(3^5)) 1]
? factorpadic(x^2 + 1, 5,3)
%2 =
[ (1 + O(5^3))*x + (2 + 5 + 2*5^2 + O(5^3)) 1]

[(1 + O(5^3))*x + (3 + 3*5 + 2*5^2 + O(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.

ffgen (*q*, *v=None*)

Return a `t_FFELT` generator for the finite field with q elements; $q = p^f$ must be a prime power. This functions computes an irreducible monic polynomial $P \in \mathbb{F}_p[X]$ of degree f (via `ffinit`) and returns $g = X(modP(X))$. If `v` is given, the variable name is used to display g, else the variable x is used.

```
? g = ffgen(8, 't);
? g.mod
%2 = t^3 + t^2 + 1
? g.p
%3 = 2
? g.f
%4 = 3
? ffgen(6)
 *** at top-level: ffgen(6)
 *** ^--------
 *** ffgen: not a prime number in ffgen: 6.
```

Alternative syntax: instead of a prime power $q = p^f$, one may input the pair $[p, f]$:

```
? g = ffgen([2,4], 't);
? g.p
%2 = 2
? g.mod
%3 = t^4 + t^3 + t^2 + t + 1
```

Finally, one may input directly the polynomial P (monic, irreducible, with `t_INTMOD` coefficients), and the function returns the generator $g = X(modP(X))$, inferring p from the coefficients of P. If `v` is given, the variable name is used to display g, else the variable of the polynomial P is used. If P is not irreducible, we create an invalid object and behaviour of functions dealing with the resulting `t_FFELT` is undefined; in fact, it is much more costly to test P for irreducibility than it would be to produce it via `ffinit` .

ffinit (*p*, *n*, *v=None*)

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` .

fflog (*x*, *g*, *o=None*)

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` .

If no o is given, assume that g is a primitive root. 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
```

ffnbirred (*q, n, fl=0*)

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=None*)

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
```

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(...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 (*x*)

Fractional part of x. Identical to $x - floor(x)$. If x is real, the result is in $[0, 1[$.

fromdigits (*x*, *b=None*)

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
```

galoisexport (*gal*, *flag=0*)

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=0*, *v=None*)

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)). Example:

```
? G = galoisinit(x^4+1);
? galoisfixedfield(G,G.group[2],2)
%2 = [x^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - y*x - 1, x^2 + y*x - 1]]
```

computes the factorization $x^4 + 1 = (x^2 - \sqrt{-2}x - 1)(x^2 + \sqrt{-2}x - 1)$

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=None*)

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 36 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

$1 = H_0 \triangleleft H_1 \triangleleft ... \triangleleft H_{n-1} \triangleleft H_n$

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 or S_4.

In practice, almost all small groups are WKSS, the exceptions having order 36(1 exception), 48(2), 56(1), 60(1), 72(5), 75(1), 80(1), 96(10) and ≥ 108.

This function is a prerequisite for most of the `galois` xxx 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 $[o_1, ..., o_g]$ ends by $[2, 2, 3]$.

* if G/H_n S_4 then $[o_1, ..., o_g]$ ends by $[2, 2, 3, 2]$.

* for $1 <= i <= g$ the subgroup of G generated by $[s_1, ..., s_g]$ is normal, with the exception of $i = g - 2$ in the A_4 case and of $i = g - 3$ in the S_A case.

* the relative order o_i of s_i is its order in the quotient group $G/ < s_1, ..., s_{i-1} >$, with the same exceptions.

* for any $x \in G$ there exists a unique family $[e_1, ..., e_g]$ such that (no exceptions):

– for $1 <= i <= g$ we have $0 <= e_i < o_i$

– $x = g_1^{e_1} g_2^{e_2} ... g_n^{e_n}$

If present den must be a suitable value for $gal[5]$.

galoisisabelian (*gal*, *flag=0*)

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 $fl = 0$, 1 if $fl = 1$.

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=None*, *fl=0*, *v=None*)

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(:math:`n)`. $H$ as in the first case above, or a matrix, taken to be a HNF left divisor of the SNF for $(\mathbb{Z}/n\mathbb{Z})^*$ (of type `:math:`N.cyc`), giving the generators of H in terms of `:math:`N.gen`.
- N the output of `bnrinit(bnfinit(y), :math:`m, 1)` 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.gen`.

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):

```
polsubcyclo(n, d = -1)=
{ my(bnr,L,IndexBound);
 IndexBound = if (d < 0, n, [d]);
 bnr = bnrinit(bnfinit(y), [n,[1]], 1);
 L = subgrouplist(bnr, IndexBound, 1);
 vector(#L,i, galoissubcyclo(bnr,L[i]));
}
```

Setting `L = subgrouplist(bnr,IndexBound)` would produce subfields of exact conductor noo.

galoissubfields (*G*, *flag=0*, *v=None*)
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=0*)
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 non-positive 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 $\Gamma_p(k) = (-1)^k \prod_{j<k}' j$, where the prime means that p does not divide j.

```
? gamma(1/4 + O(5^10))
%1= 1 + 4*5 + 3*5^4 + 5^6 + 5^7 + 4*5^9 + O(5^10)
? algdep(%,4)
%2 = x^4 + 4*x^2 + 5
```

gammah (*x*, *precision=0*)
Gamma function evaluated at the argument $x + 1/2$.

gammamellininv (*G*, *t*, *m=0*, *precision=0*)
Returns the value at t of the inverse Mellin transform G initialized by `gammamellininvinit` .

```
? G = gammamellininvinit([0]);
? gammamellininv(G, 2) - 2*exp(-Pi*2^2)
%2 = -4.484155085839414627 E-44
```

The alternative shortcut

```
gammamellininv(A,t,m)
```

for

```
gammamellininv(gammamellininvinit(A,m), t)
```

is available.

gammamellininvasymp (*A*, *serprec=-1*, *n=0*)
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

$$K^{(m)}(t) = \sqrt{2^{d+1}/d}t^{a+m(2/d-1)}e^{-d\pi t^{2/d}} \sum_{n>=0} M[n + 1](\pi t^{2/d})^{-n}$$

with $a = (1 - d + \sum_{1<=j<=d} a_j)/d$.

gammamellininvinit (*A*, *m=0*, *precision=0*)
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.

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=None*)

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 `bezout` 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 + O(y); gcd(b,b)
%2 = y
? c = 4*x + O(2^3); gcd(c,c)
%3 = 4
```

A good quantitative check to decide whether such a gcd "should be" non-trivial, is to use `polresultant` : a value close to 0 means that a small deformation of the inputs has non-trivial 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=None*)

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 E_1 and E_2 intersecting transversally at one point; *vecj* contains their modular invariants j_1 and j_2, which may live in a quadratic extension of $\mathbb{F}_p$ and need not be distinct. The Jacobian $J(C)$ has potentially good reduction, isomorphic to the product of the reductions of E_1 and E_2.
- 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'-α]‘ in the notation of Namikawa-Ueno.
- The figure `[2I_0-m]` in Namikawa-Ueno, page 159, must be denoted by `[2I_0-(m+1)]`.

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 non-zero 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=None*)
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 non-singular 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*, *p*, *n*)
Let X be the curve defined by $y^2 = Q(x)$, where Q is a polynomial of degree d over $\mathbb{Q}$ and $p >= d$ a prime such that X has good reduction at p 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 p. The matrix is computed to absolute p-adic precision p^n.

hyperu (*a*, *b*, *x*, *precision=0*)
U-confluent hypergeometric function with parameters a and b. The parameters a and b can be complex but the present implementation requires x to be positive.

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: non-square t_MAT in idealtyp.
? c = [2, 0; 2, 0]; idealadd(K, c, c) \\ non-sense
%6 =
[2 0]

[0 2]
? d = [1, 0; 0, 2]; idealadd(K, d, d) \\ non-sense
%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=None*)

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 $1 <= i <= k$ and $\sum_{1<=i<=k} e[i] = 1$.

idealappr (*nf*, *x*, *flag=0*)

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 non-zero, we have $v_p(\alpha) = v_p(x)$, and $v_p(\alpha) >= 0$ for all other p.

If $flag$ is non-zero, x must be given as a prime ideal factorization, as output by `idealfactor` , but possibly with zero or negative exponents. This yields an element α such that for all prime ideals p occurring in x, $v_p(\alpha)$ is equal to the exponent of p in x, and for all other prime ideals, $v_p(\alpha) >= 0$. This generalizes $idealappr(nf, x, 0)$ since zero exponents are allowed. Note that the algorithm used is slightly different, so that

```
idealappr(nf, idealfactor(nf,x))
```

may not be the same as `idealappr(nf,x,1)` .

idealchinese (*nf*, *x*, *y=None*)

x being a prime ideal factorization (i.e. a 2 by 2 matrix whose first column contains prime ideals, and the second column integral exponents), y a vector of elements in nf indexed by the ideals in x, computes an element b such that

$v_p(b - y_p) >= v_p(x)$ for all prime ideals in x and $v_p(b) >= 0$ for all other p.

```
? K = nfinit(t^2-2);
? x = idealfactor(K, 2^2*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 $s_i sign(\sigma_i(b)) >= 0$ for all i. In other words, the sign is fixed to s_i at the i-th embedding whenever s_i is non-zero.

```
? 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 $\beta.x$ is an integral ideal coprime to y.

idealdiv (*nf*, *x*, *y*, *flag=0*)
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 non-zero, 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.

idealfactor (*nf*, *x*)
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.

idealfactorback (*nf*, *f*, *e=None*, *flag=0*)
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 non-zero, 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 $\mathrm{Frob}(x) = x^p mod p$ 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 $101 = 8 mod 31$.

idealhnf (*nf*, *u*, *v=None*)

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^3 - 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 $\mathbb{Z}$-basis of the ideal, in particular that the matrix has maximal rank N. This acts as `mathnf` since the $\mathbb{Z}_K$-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 $a\mathbb{Z} + (1/2)(-b + \sqrt{D_K})\mathbb{Z}$.

```
? K = nfinit(x^2 - 60); K.disc
%1 = 60
? idealhnf(K, qfbprimeform(60,2))
%2 =
[2 1]

[0 1]
? idealhnf(K, Qfb(1,2,3))
 *** at top-level: idealhnf(K,Qfb(1,2,3
 *** ^--------------------
 *** idealhnf: Qfb(1, 2, 3) has discriminant != 60 in idealhnf.
```

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 $[I^{-1}, 1/t]$.

ideallist (*nf*, *bound*, *flag=4*)

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 `bnf.tufu` . 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, l[i].clgp)
%3 = [[20, [20]], [16, [4, 4]], [20, [20]]]
? l[1].mod
%4 = [[25, 18; 0, 1], []]
? l[2].mod
%5 = [[5, 0; 0, 5], []]
? l[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(#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 the ray class group. 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 a non-necessarily integral element of *nf* which must have valuation equal to 0 at all prime ideals in the support of D. 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 $0 <= x_i < d_i$ 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 `idealstar(,N)`.

idealmin (*nf*, *ix*, *vdir=None*)
This function is useless and kept for backward compatibility only, use :literal:‘idealred‘. Computes a pseudo-minimum of the ideal x in the direction *vdir* in the number field *nf*.

idealmul (*nf*, *x*, *y*, *flag=0*)
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 non-zero, 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=0*)
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 non-zero, 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=0*)
Computes the prime ideal decomposition of the (positive) prime number p in the number field K represented by *nf*. If a non-prime p is given the result is undefined. If f is present and non-zero, restrict the result to primes of residue degree $<= f$.

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 and 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's 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 follow:

- `g[1]` is the decomposition group of P,
- `g[2]` is $G_0(P)$, the inertia group of P,

and for $i >= 2$,

•`g[i]` is $G_{i-2}(P)$, the $i-2$-th ramification group of P.

The length of g is the number of non-trivial 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=None*)

LLL reduction of the ideal I in the number field *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 *non-negative* 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 a "small" a in I (see the end for technical details). The result is the Hermite normal form of the "reduced" ideal $J = rI/a$, where r is the unique rational number such that J is integral and primitive. (This is usually not a reduced ideal in the sense of Buchmann.)

```
? K = nfinit(y^2+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 non-trivial result does not prove that the ideal is non-principal. 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 $[J, sa/r]$; this way, one can keep 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 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 and is "expected to be small".

idealstar (*nf*, *N*, *flag=1*)

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` .

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 much faster.

idealtwoelt (*nf*, *x*, *a=None*)

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.
- When called as `idealtwoelt(nf,x,a)` with an explicit non-zero 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 much slower than the default variant.

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 `+oo` .

imag (*x*)

Imaginary part of x. When x is a quadratic number, this is the coefficient of ω in the "canonical" integral basis $(1, \omega)$.

incgam (*s*, *x*, *g=None*, *precision=0*)
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=0*)
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 (*x*, *v=None*)
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 (*a*, *b*, *m=0*, *precision=0*)
Initialize tables for integration from a to b, where a and b are coded as in `intnum` . 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 `intnuminit(-1,1)` is equivalent to `intnuminit(0,Pi)` , and `intnuminit([0,-1/2],oo)` is equivalent to `intnuminit([-1,-1/2],-oo)` ; on the other hand, the order matters and `intnuminit([0,-1/2],[1,-1/3])` is *not* equivalent to `intnuminit([0,-1/3],[1,-1/2])` !

If m is 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 `intnum` 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 when non-default behaviour is specified at end points, `intnuminit` 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 behaviour of F at endpoints, depending on z.

isfundamental (*x*)

True (1) if x is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise.

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=0*)

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 prime and has more than 1000 digits, say. Use `ispseudoprime` to quickly check for compositeness. See also `factor`. It accepts vector/matrices arguments, and is then applied componentwise.

If $flag = 0$, use a combination of Baillie-PSW pseudo primality test (see `ispseudoprime`), Selfridge "$p - 1$" test if $x - 1$ is smooth enough, and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) for general x.

If $flag = 1$, use Selfridge-Pocklington-Lehmer "$p - 1$" test and output a primality certificate as follows: return

- 0 if x is composite,
- 1 if x is small enough that passing Baillie-PSW test guarantees its primality (currently $x < 2^{64}$, as checked by Jan Feitsma),
- 2 if x is a large prime whose primality could only sensibly be proven (given the algorithms implemented in PARI) using the APRCL test.
- Otherwise (x is large and $x - 1$ is smooth) output a three column matrix as a primality certificate. The first column contains prime divisors p of $x - 1$ (such that $\prod p^{v_p(x-1)} > x^{1/3}$), the second the corresponding elements a_p as in Proposition 8.3.1 in GTM 138 , and the third the output of isprime(p,1).

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 `isprime` at a high enough debug level, you may see warnings about untested integers being declared primes. This is normal: we ask for partial factorisations (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 `isprime` procedure.

If $flag = 2$, use APRCL.

ispseudoprime (*x*, *flag=0*)

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 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).

There are no known composite numbers passing this test, although it is expected that infinitely many such numbers exist. In particular, all composites $<= 2^{64}$ 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).

issquarefree (*x*)

True (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial.

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.
- $(x\| - 1) = 1$ if $x >= 0$ 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=0*)

Lambert W function, solution of the implicit equation $xe^x = y$, for $y > 0$.

lcm (*x*, *y=None*)

Least common multiple of x and y, i.e. such that $\mathrm{lcm}(x, y) * \gcd(x, y) = x * y$, 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 non-negative 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 `l` 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),
- 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
```

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. 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
```

lfun (*L*, *s*, *D=0*, *precision=0*)

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 $B = realbitprecision$.

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+O(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 + O(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=0*, *precision=0*)

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=0*)

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, 1]
? lfunan(5, 10) \\ Dirichlet L-function for kronecker(5,.)
%2 = [1, -1, -1, 1, 0, 1, -1, -1, 1, 0]
```

`lfunartin` (*nf*, *gal*, *M*, *n*)

Returns the `Ldata` structure attached to the Artin L-function attached to 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 M is the vector of the image of the generators `:emphasis:`gal` .gen‘ by ρ. The elements of M are matrices with polynomial entries, 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);
? N = nfinit(nfsplitting(P));
? G = galoisinit(N);
? L1 = lfunartin(N,G, [[a,0;0,a^-1],[0,1;1,0]], 5);
? L2 = lfunartin(N,G, [[a^2,0;0,a^-2],[0,1;1,0]], 5);
? s = 1 + x + O(x^4);
? lfun(1,s)*lfun(-47,s)*lfun(L1,s)^2*lfun(L2,s)^2 - lfun(N,s)
%6 ~ 0
? lfun(1,s)*lfun(L1,s)*lfun(L2,s) - lfun(P,s)
%7 ~ 0
? bnr = bnrinit(bnfinit(x^2+47),1,1);
? lfun([bnr,[1]], s) - lfun(L1, s)
%9 ~ 0
? lfun([bnr,[1]], s) - lfun(L1, s)
%10 ~ 0
```

The first identity is the factorisation of the regular representation of D_{10}, the second the factorisation of the natural representation of $D_{10} \subset S_5$, the next two are the expressions of the degree 2 representations as induced from degree 1 representations.

lfuncheckfeq (*L*, *t=None*, *precision=0*)

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} is a computed value derived from $\overline{\theta}(t)$ and $\theta(1/t)$ at some $t! = 0$ and the assumed functional equation. Of course, a large negative value of the order of `realbitprecision` is expected.

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.

```
? \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*, *ab=None*, *flag=0*, *precision=0*)

Compute the conductor of the given L-function (if the structure contains a conductor, it is ignored); $ab = [a, b]$ is the interval where we expect to find the conductor; it may be given as a single scalar b, in which case we look in $[1, b]$. Increasing `ab` slows down the program but gives better accuracy for the result.

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, 857.0000000000000000]

? \p38
realprecision = 38 significant digits
? lfunconductor(L)
%4 = 857
```

Note. This program should only be used if the primes dividing the conductor are unknown, which is rare. If they are known, a direct search through possible prime exponents using `lfuncheckfeq` will be more efficient and rigorous:

```
? E = ellinit([0,0,0,4,0]); /* Elliptic curve y^2 = x^3+4x */
? E.disc \\ |disc E| = 2^12
%2 = -4096
\\ create Ldata by hand. Guess that root number is 1 and conductor N
? L(N) = lfuncreate([n->ellan(E,n), 0, [0,1], 1, N, 1]);
? fordiv(E.disc, d, print(d,": ",lfuncheckfeq(L(d))))
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
? lfunconductor(L(1)) \\ lfunconductor ignores conductor = 1 in Ldata !
%5 = 32
```

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(N) = lfuncreate([n->ellan(E,n), 0, [0,1], 1, N, -1]);
? [ lfuncheckfeq(L2(d)) | d<-divisors(E.disc) ]
%7 = [0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, -1, -1]
```

lfuncost (*L*, *sdom=None*, *der=0*, *precision=0*)

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 fields is the product of four Dirichlet L-functions attached to the trivial character, a non-trivial real character and two complex conjugate characters. The non-trivial 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. You are urged to use a high-level constructor when one is available, and this function accepts them, see `??lfun` :

```
? 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 $\Re(s-c) <= w, \Im(s) <= h$.

We now describe the low-level interface, used to input non-builtin 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. 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.

A second format is limited to multiplicative L functions affording an Euler product. It is a closure of arity 2 `(p,d)-> L(p)` giving the local factor L_p at p as a rational function, to be evaluated at p^{-s} as in `direuler`; d is set to the floor of $\log_p(n)$, where n is the total number of Dirichlet coefficients $(a_1, ..., a_n)$ that will be computed in this way. This parameter d allows to compute only part of L_p when p is large and L_p expensive to compute, but it can of course be ignored by the closure.

Finally one can describe separately the generic Dirichlet coefficients and the bad local factors by setting $dir = [an, [p_1, L_{p_1}^{-1}], ..., [p_k, L_{p_k}^{-1}]]$, where `an` describes the generic coefficients in one of the two formats above, except that coefficients a_n with $p_i \| n$ for some $i <= k$ will be ignored. The subsequent pairs $[p, L_p^{-1}]$ give the bad primes and corresponding *inverse* local factors.

- `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

$where : math : `\Gamma_{\mathbb{R}}(s) = \pi^{-s/2}\Gamma(s/2)`.Thissamesyntaxisusedinthe : literal : `gammamellininv`functions.Inpar$

- `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 $w = k - 1$.
- `V[5] = N` is the conductor, an integer $N >= 1$, 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 $[\beta, P_\beta(x)]$, where β is a pole and the power series $P_\beta(x)$ describes the attached polar part, such that $L(s) - P_\beta(s-\beta)$ is holomorphic in a neighbourhood of β. For instance $P_\beta = r/x + O(1)$ for a simple pole at β or $r_1/x^2 + r_2/x + O(1)$ 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.

The latter is mandatory unless $a = \overline{a^*}$ (coded by `astar` equal to 0 or 1): otherwise, the poles of L^* cannot be infered from the poles of L ! (Whereas the functional equation allows to deduce the polar part of Λ^* from the polar part of Λ.) The special coding $poles = r$ a complex scalar is available in this case, to describe a L function with at most a single simple pole at $s = k$ and residue r. (This is the usual situation, for instance for Dedekind zeta functions.) This value r can be set to 0 if unknown, and it will be computed.

lfundiv (*L1*, *L2*, *precision=0*)

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 $v_z(L_1) >= v_z(L_2)$ at all complex numbers z: the construction may not create new poles, nor increase the order of existing ones.

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)
```

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=0*)

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: if as usual $k/2$ is the center of the critical strip, d is the degree, α_j the entries of `Vga` giving the gamma factors, and ε the root number, then if we set $s = k/2 + it = \rho e^{i\theta}$ and $E = (d(k/2-1) + \sum_{1<=j<=d} \alpha_j)/2$, the computed function at t is equal to

$$Z(t) = \varepsilon^{-1/2}\Lambda(s).\|s\|^{-E}e^{dt\theta/2},$$

which is a real function of t for self-dual Λ, 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.

```
? 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))
```

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=0*, *precision=0*)

Initalization 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, $\|\Re(s) - center\| <= w$, $\|\Im(s)\| <= h$, where all three components of `sdom` are real and w, h are non-negative. `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=0*, *precision=0*)

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 $B = realbitprecision$.

lfunmfspec (*L*, *precision=0*)
Returns `[valeven,valodd,omminus,omplus]` , where `valeven` (resp., `valodd`) is the vector of even (resp., odd) periods of the modular form given by `L` , and `omminus` and `omplus` the corresponding real numbers ω^- and ω^+ normalized in a noncanonical way. For the moment, only for modular forms of even weight.

lfunmul (*L1*, *L2*, *precision=0*)
Creates the `Ldata` structure (without initialization) corresponding to the product of the Dirichlet series given by `L1` and `L2` .

lfunorderzero (*L*, *m=-1*, *precision=0*)
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 non-negative 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=0*)
Returns the `Ldata` structure attached to the Θ function of the lattice attached 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
```

lfunrootres (*data*, *precision=0*)
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 $[r, R, w]$, where r is the residue of $L(s)$ at the unique pole, R is the residue of $\Lambda(s)$, 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).

lfuntheta (*data*, *t*, *m=0*, *precision=0*)
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 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 an easy theorem on modular forms implies that $a(0)$ can be recovered by the formula $a(0) = -L(f, 0)$.

lfunthetacost (*L*, *tdom=None*, *m=0*, *precision=0*)

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); \\ 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| >= 2, |arg(t)| < 1/10
%4 = 8

? L = lfuncreate( ellinit([1,1]) );
? lfunthetacost(L) \\ for t >= 1
%6 = 2471
```

lfunthetainit (*L*, *tdom=None*, *m=0*, *precision=0*)

Initalization function for evaluating the m-th derivative of theta functions with argument t in domain *tdom*. By default (*tdom* omitted), t is real, $t >= 1$. Otherwise, *tdom* may be

- a positive real scalar ρ: t is real, $t >= \rho$.
- a non-real complex number: compute at this particular t; this allows to compute $\theta(z)$ for any complex z satisfying $\|z\| >= \|t\|$ and $\|\arg z\| <= \|\arg t\|$; we must have $\|2\arg z/d\| < \pi/2$, where d is the degree of the Γ factor.
- a pair $[\rho, \alpha]$: assume that $\|t\| >= \rho$ and $\|\arg t\| \leq \alpha$; we must have $\|2\alpha/d\| < \pi/2$, 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)])
```

lfunzeros (*L*, *lim*, *divz=8*, *precision=0*)

`lim` being either a positive upper limit or a non-empty real interval inside $[0, +oo[$, 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 $T >= 1$ 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
```

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.)

Should you pass an `Linit` argument to the function, beware that the algorithm needs at least

```
L = lfuninit(Ldata, T+1)
```

where T is the upper bound of the interval defined by `lim` : this allows to detect zeros near T. Make sure that your `Linit` domain contains this one. 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=None*)

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 + O(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 + O(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 + O(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 + O(3)) + Mod(2,3))
%1 = (1 + O(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
```

lindep (*v*, *flag=0*)

finds a small non-trivial 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 + 3^2 + 3^3 + 3^4 + O(3^5)])
%2 = [1, -2]~
```

If v is a matrix, $flag$ is ignored and the function returns a non trivial kernel vector (combination of the columns).

```
? lindep([1,2,3;4,5,6;7,8,9])
%3 = [1, -2, 1]~
```

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 = [t^2+O(t^4), 1+O(t^2)]; L=lindep(v)
%1 = [1, 0]~
? v*L
%2 = t^2+O(t^4) \\ small but not 0
```

listinsert (*L*, *x*, *n*)
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.

listpop (*list*, *n=0*)
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 $O(\#L - n + 1)$.

listput (*list*, *x*, *n=0*)
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])
```

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
```

```
 *** ^------
 *** non-existent 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 (*L*, *flag=0*)

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.

```
? 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
```

This is faster than the `vecsort` command since the list is sorted in place: no copy is made. No value returned.

If $flag$ is non-zero, suppresses all repeated coefficients.

lngamma (*x*, *precision=0*)

Principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with non-positive 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+O(x^2))
%1 = -0.57721566490153286060651209008240243104*x + O(x^2)
? lngamma(x+O(x^2))
 *** at top-level: lngamma(x+O(x^2))
 *** ^-----------------
 *** lngamma: domain error in lngamma: valuation != 0
? lngamma(-1+x+O(x^2))
 *** lngamma: Warning: normalizing a series with 0 leading term.
 *** at top-level: lngamma(-1+x+O(x^2))
 *** ^--------------------
 *** lngamma: domain error in intformal: residue(series, pole) != 0
```

log (*x*, *precision=0*)

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.

mapdelete (*M*, *x*)

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: non-existent component in mapget: index not in map
```

mapput (*M*, *x*, *y*)

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=0*)

adjoint matrix of M, i.e. a matrix N of cofactors of M, satisfying $M * N = \det(M) * \mathrm{Id}$. M must be a (non-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);
%2 =
[Mod(1, 4) Mod(2, 4) Mod(3, 4)]

[Mod(3, 4) Mod(0, 4) Mod(1, 4)]
```

```
[Mod(2, 4) Mod(3, 4) Mod(0, 4)]
```

Both algorithms use $O(n^4)$ operations in the base ring, and are usually slower than computing the characteristic polynomial or the inverse of M directly.

matalgtobasis (*nf*, *x*)
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*)
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 non-zero 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]

[3 4 0 0 0]

[0 0 1 2 3]

[0 0 4 5 6]

[0 0 7 8 9]
```

matdet (*x*, *flag=0*)

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 non-zero 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 $\det(B^T B)$ 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 and the matrix non-square, 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.

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=0*, *precision=0*)

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]
```

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=0*, *v=None*)
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).

mathnf (*M*, *flag=0*)
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 $0, 1, 2, 3$ of $flag$ have a binary meaning, analogous to the one in `matsnf`; in this case, binary digits of $flag$ mean:

- 1 (complete output): if set, outputs $[H, U]$, 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.
- 2 (generic input): *Deprecated*. If set, assume that $R = K[X]$ is a polynomial ring; otherwise, assume that $R = \mathbb{Z}$. This flag is now useless since the routine always checks whether the matrix has integral entries.

For these 4 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 in general 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 (x, d)

If x is a (not necessarily square) matrix of maximal rank with integer entries, and d is a multiple of the (non-zero) 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]

[ 0 0 1]
```

mathouseholder (*Q*, *v*)
applies a sequence Q of Householder transforms, as returned by `matqr` $(M, 1)$ to the vector or matrix v.

matimage (*x*, *flag=0*)
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.

matindexrank (*x*)
x 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 $vecextract(x, y, z)$ is invertible.

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 $\mathbb{Q}$-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 much 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]
```

matisdiagonal (*x*)
Returns true (1) if x is a diagonal matrix, false (0) if not.

matker (*x*, *flag=0*)
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=0*)
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.

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.

matqr (*M*, *flag=0*, *precision=0*)
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.

matrixqz (*A*, *p=None*)
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 $p >= 0$, A is assumed to have maximal rank $n <= m$. 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.

```
? 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] \\ determinants of all 2x2 minors
? B = matrixqz(A)
%2 =
[3 1]

[5 2]

[7 3]
? minors(%)
%3 = [1, 2, 1] \\ B integral with coprime minors
```

If $p = -1$, returns the HNF basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{Z}}A$.

If $p = -2$, returns the HNF basis of the lattice $\mathbb{Z}^n \cap Im_{\mathbb{Q}}A$.

```
? matrixqz(A,-1)
%4 =
[8 5]

[4 3]

[0 1]

? matrixqz(A,-2)
%5 =
[2 -1]

[1 0]

[0 1]
```

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=0*)
If X is a (singular or non-singular) matrix outputs the vector of elementary divisors of X, i.e. the diagonal of the Smith normal form of X, normalized so that $d_n \| d_{n-1} \| ... \| d_1$.

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.

2 (generic input): if set, allows polynomial entries, in which case the input matrix must be square. Otherwise, assume that X has integer coefficients with arbitrary shape.

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*)

M being an invertible matrix and B a column vector, finds the solution X of $MX = B$, using Dixon p-adic lifting method if M and B are integral and Gaussian elimination otherwise. This has the same effect as, but is faster, than $M^{-1} * B$.

matsolvemod (*M*, *D*, *B*, *flag=0*)

M being any integral matrix, D a column vector of non-negative integer moduli, and B an integral column vector, gives a small 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 = [-2, 0]~
? matsolvemod(M, 3, 1) \\ M X = [1,1]~ over F_3
%3 = [-1, 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 a small 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:

```
? matsupplement(matrix(2,0))
 *** at top-level: matsupplement(matrix
 *** ^--------------------
 *** matsupplement: sorry, suppl [empty matrix] is not yet implemented.
```

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.

min (*x*, *y*)

Creates the maximum of x and y when they can be compared.

minpoly (*A*, *v=None*)

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, x^3 - x -1); v = u^5;
? w = modreverse(v)
%2 = Mod(x^2 - 4*x + 1, x^3 - 5*x^2 + 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 of type integer.

msatkinlehner (*M, Q, H=None*)
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
```

mscuspidal (*M, flag=0*)
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]
```

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 -3]

[1 0]

? E == mscuspidal(M,1)[2]
%4 = 1
```

mseval (*M*, *s*, *p=None*)

Let $\Delta := Div^0(\mathbb{P}^1(\mathbb{Q}))$. 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 $\mathrm{Hom}_G(\Delta, V)$, and let $p = [a, b] \in \Delta$ 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 $oo = (1 : 0)$. The symbol s is either

- a `t_COL` coding an element of a modular symbol subspace in terms of the fixed basis of $\mathrm{Hom}_G(\Delta, V)$ chosen in M; if M was initialized with a non-zero *sign* ($+$ or $-$), then either the basis for the full symbol space or the $\pm$-part can be used (the dimension being used to distinguish the two).
- 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, 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 the symbol s to the second form: a vector of the $s(g_i)$.

```
? 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.

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 $\mathbb{P}^1(\mathbb{Q})/G$ can be given either as oo $(= (1:0))$, 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 $\gamma_r.r = (1:0)$. These symbol 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]~
? mseval(M, s)
%5 = [1, 0]
? s = msfromcusp(M,1)
%6 = [-5/16, -15/32, -3/32, 0]~
? 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 non-zero *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=0*)

Let $E/\mathbb{Q}$ be an elliptic curve of conductor N. For $\varepsilon = \pm 1$, 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 $dx/(2y + a_1x + a3)$ 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_oo is the number of connected components of $E(\mathbb{R})$, Ω^- is equal to $(-2/c_oo)ximag(E.omega[2])$. 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 $x^\pm$ 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 $2/c_oox$ 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 $sign = \pm 1$, and $x = [x^+, x^-]$ when *sign* is 0. The modular symbols $x^\pm$ are given as a `t_COL` (in terms of the fixed basis of $\mathrm{Hom}_G(\Delta, \mathbb{Q})$ 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 \\ both x^+ and x^-
%5 = [[1/5, -1/2, -1/2]~, [0, 1/2, -1/2]~]
? 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]
```

msfromhecke (*M*, *v*, *H=None*)

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*)
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=None*)
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=0*)

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 $\mathrm{Hom}_G(Div^0(\mathbb{P}^1(\mathbb{Q})), V)$. This is canonically isomorphic to $H^1_c(X(G), V)$, and allows to compute modular forms for G. If *sign* is present and non-zero, 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 $N > 1$. The only supported representation is $V_k = \mathbb{Q}[X,Y]_{k-2}$, coded by the integer $k >= 2$.

msissymbol (*M*, *s*)

M being a full modular symbol space, as given by `msinit` , check whether s is a modular symbol attached to M.

```
? M = msinit(7,8, 1); \\ M_8(Gamma_0(7))^+
? N = msnew(M)[1];
? s = N[,1];
? msissymbol(M, s)
%4 = 1
? S = mseval(M,s);
? msissymbol(M, S)
%6 = 1
? [g,R] = mspathgens(M); g
%7 = [[+oo, 0], [0, 1/2], [1/2, 1]]
? #R \\ 3 relations among the generators g_i
%8 = 3
? T = S; T[3]++; \\ randomly perturb S(g_3)
? msissymbol(M, T)
%10 = 0 \\ no longer satisfies the relations
```

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=None*, *r=0*)

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 $< \chi >$:

$$L_p^{(r)}(\Phi, \chi^s) = \int_{\mathbb{Z}_p^*} \chi^s(z)(\log z)^r d\mu(z).$$

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 $s_2 mod p - 1$ or $s_2 mod 2$ for $p = 2$, encoding the p-adic character $\chi^s := < \chi >^{s_1} \tau^{s_2}$; here χ is the cyclotomic character from $Gal(\mathbb{Q}_p(\mu_{p^o o})/\mathbb{Q}_p)$ 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 + ...
? mspadicL(mu,1) \\ L_p(chi), zero for parity reasons
%6 = [O(5^13)]~
? mspadicL(mu,2) \\ L_p(chi^2)
%7 = 3 + 4*5 + 4*5^2 + 3*5^5 + ...
? mspadicL(mu,[0,2]) \\ L_p(tau^2)
%8 = 3 + 5 + 2*5^2 + 2*5^3 + ...
? 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 + O(5^13)
? mspadicL(mu,0,1) \\ L_p'(chi^0)
%10 = 2*5 + 4*5^2 + 3*5^3 + ...
? mspadicL(mu, 2, 1) \\ L_p'(chi^2)
%11 = 4*5 + 3*5^2 + 5^3 + 5^4 + ...
```

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 = O(5^13) \\ always 0, phi is in the + part and D < 0
```

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);
? mu = mspadicmoments(Mp, phi);
? mspadicL(mu)
%7 = 1 + 3 + 3^3 + 3^4 + 2*3^5 + 3^6 + O(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 (*M*, *p*, *n*, *flag=-1*)

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 $p \| a_p$, 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=1*)

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=0*)

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

$$^L_p(\mu,\tau^i)(x) = \int_{\mathbb{Z}_p^*} \tau^i(t)(1+x)^{\log_p(t)/\log_p(u)} d\mu(t)$$

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 $G_oo := Gal(\mathbb{Q}(\mu_{p^{oo}})/\mathbb{Q})$, let $\chi : G_oo \to \mathbb{Z}_p^*$ be the cyclotomic character (isomorphism) and γ the element of G_oo such that $\chi(\gamma) = u$; then $\chi(\gamma)^{\log_p(t)/\log_p(u)} =< t >$.

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 + O(5^9)) \
 + (3 + 3*5 + 5^2 + 5^3 + 2*5^4 + 5^6 + O(5^7))*x \
```

```
 + (2 + 3*5 + 5^2 + 4*5^3 + 2*5^4 + O(5^5))*x^2 \
 + (3 + 4*5 + 4*5^2 + O(5^3))*x^3 \
 + (3 + O(5))*x^4 + O(x^5)
```

An example with non-zero 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 + O(3^12)) \
 + (1 + 3 + 2*3^2 + 3^3 + 3^5 + 2*3^6 + 2*3^8 + O(3^9))*x \
 + (1 + 2*3 + 3^4 + 2*3^5 + O(3^6))*x^2 \
 + (3 + O(3^2))*x^3 + O(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 + O(3^7)) \
 + (2 + 3^3 + O(3^5))*x \
 + (1 + 2*3 + O(3^2))*x^2 + O(x^3),\
 (3^-1 + 1 + 3 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + O(3^7)) \
 + (1 + 2*3 + 2*3^2 + 3^3 + 2*3^4 + O(3^5))*x \
 + (3^-2 + 3^-1 + O(3^2))*x^2 + O(3^-2)*x^3 + O(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 + O(3^12)) \
 + (2*3^2 + 2*3^6 + 3^7 + 3^8 + O(3^9))*x \
 + (3^3 + O(3^6))*x^2 + O(3^2)*x^3 + O(x^4)
? mspadicL(mu)
%6 = [2*3^2 + 2*3^4 + 3^5 + 3^6 + 2*3^7 + 2*3^10 + O(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 + O(3^10)
? mspadicseries(mu,1) \\ must be 0
%8 = O(3^12) + O(3^9)*x + O(3^6)*x^2 + O(3^2)*x^3 + O(x^4)
```

mspathgens (*M*)

Let $\Delta := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by `msinit` , return a set of $\mathbb{Z}[G]$-generators for Δ. 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 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 coded by a "factorization matrix": the first column contains distinct elements of G, and the second integers:

```
? 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 $\Delta := Div^0(\mathbb{P}^1(\mathbb{Q}))$. Let M being a full modular symbol space, as given by `msinit` , encoding fixed $\mathbb{Z}[G]$-generators (g_i) of Δ (see `mspathgens`). A path $p = [a, b]$ between two elements in $\mathbb{P}^1(\mathbb{Q})$ corresponds to $[b] - [a] \in \Delta$. The path extremities a and b may be given as `t_INT` , `t_FRAC` or $oo = (1:0)$.

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]
```

Note that the output depends only on the group G, not on the representation V.

msqexpansion (*M*, *projH*, *serprec=-1*)

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, $n <= B$ of the corresponding normalized newform. If B is omitted, use `seriesprecision` .

This function uses a naive $O(B^2 d^3)$ 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);
? msqexpansion(M,s,10)
%3 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
? ellan(E, 10)
%4 = [1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
```

mssplit (*M*, *H*, *dimlim=0*)

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). This function split H into Hecke-simple subspaces. If `dimlim` is present and positive, restrict to subspaces of dimension $<= dimlim$. 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, 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=None*)

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, $v_p(N) <= 1$, and weight k which is an eigenvector for the Hecke operator $T_N(p)$ for a non-zero 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 $p \nmid N$ 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 $p \nmid N$ 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] \\ 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 (*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 `+oo`.

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(y^2 + 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, y^2+4))
%5 = [0, 2]~
```

This is the inverse function of `nfbasistoalg`.

nfbasistoalg (*nf*, *x*)

Given an algebraic number x in the number field *nf*, transforms it into `t_POLMOD` form.

```
? nf = nfinit(y^2 + 4);
? nf.zk
%2 = [1, 1/2*y]
? nfbasistoalg(nf, [1,1]~)
%3 = Mod(1/2*y + 1, y^2 + 4)
? nfbasistoalg(nf, y)
%4 = Mod(y, y^2 + 4)
? nfbasistoalg(nf, Mod(y, y^2+4))
%5 = Mod(y, y^2 + 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=0*)

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) \\ 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 \\ a fifth root of y
%5 = 0
? [T, X] = rnfpolredbest(K, R, 1);
? lift(T) \\ simpler defining polynomial for K[x]/(R)
%7 = x^10 + (-11/2*y + 25/2)
? liftall(X) \\  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); \\ 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 non-zero 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
```

nfeltadd (*nf*, *x*, *y*)

Given two elements x and y in *nf*, computes their sum $x + y$ in the number field nf.

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*)
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.

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*)
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*)
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*)
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*.

nfelttrace (*nf*, *x*)
Returns the absolute trace of x.

nffactor (*nf*, *T*)
Factorization of the univariate polynomial 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^2 + 1);
? nffactor(nf, x^2 + y); \\ OK
```

```
? nffactor(nf, x^2 + Mod(y, y^2+1)); \\  OK
? nffactor(nf, x^2 + Mod(z, z^2+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 compute the field discriminant due to integer factorization difficulties. In the latter case, if you must use a partial discriminant factorization (as allowed by both `nfdisc` or `nfbasis`) to build a partially correct `nf` structure, always input `nf.pol` to `nffactor` , and not your makeshift *nf*: otherwise factors could be missed.

nffactorback (*nf*, *f*, *e=None*)

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^2+1);
? nffactorback(nf, [3, y+1, [1,2]~], [1, 2, 3])
%2 = [12, -66]~
? 3 * (I+1)^2 * (1+2*I)^3
%3 = 12 - 66*I
```

nffactormod (*nf*, *Q*, *pr*)

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) `modprinit` 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 (*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 non-trivial 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);
? l = nfgaloisconj(nf); aut = l[2] \\  automorphisms in basistoalg form
%2 = -31/11*x^2 + 1109/11*x - 925/11
? L = matalgtobasis(nf, l); AUT = L[2] \\  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 = 1,451 ms.
? for (i=1,10^5, nfgaloisapply(nf, AUT, v))
time = 1,045 ms. \\  but the latter is faster
```

nfgaloisconj (*nf*, *flag=0*, *d=None*, *precision=0*)

nf being a number field as output by `nfinit`, computes the conjugates of a root r of the non-constant 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 $flag = 2$ (OBSOLETE), use complex approximations to the roots and an integral LLL. The result is not guaranteed to be complete: some conjugates may be missing (a warning is issued if the result is not proved complete), especially so if the corresponding polynomial has a huge index, and increasing the default precision may help. This variant is slow and unreliable: don't use it.

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=None*)

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 behaviour 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, 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 (*nf*, *a*, *b*, *pr=None*)

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=0*)

Given a pseudo-matrix (A, I), finds a pseudo-basis (B, J) in Hermite normal form of the module it generates. If $flag$ is non-zero, 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=0*, *precision=0*)

pol being a non-constant, preferably monic, irreducible polynomial in $\mathbb{Z}[X]$, 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 `nf`*xxx* 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 specified 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 useful for certain computations in the number field K.

* 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.

* Finally, 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). $MDI$ is only used in `idealinv`.

$nf[6]$ 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 an integral basis for $\mathbb{Z}_K$ (`:emphasis:`nf` .zk‘) expressed on the powers of θ. 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 polynomial is input, `nfinit` will transform it into a monic one, then reduce it (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 special input format $[pol, B]$ is also accepted where *pol* is a polynomial as above and B has one of the following forms

- an integer basis, as would be computed by `nfbasis` : a vector of polynomials with first element 1. This is useful if the maximal order is known in advance.
- an argument `listP` which specifies a list of primes (see `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 is unconditionaly correct when the discriminant `nf.disc` factors completely over this set of primes. 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 $<= 10^3$ (no prime $<= 10^3$ divides `nf.index`). The certification step proves the correctness of the computation.

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: first a linear change of variables is performed, to get a monic polynomial, then `polredbest` .

nfisideal (*nf*, *x*)
Returns 1 if x is an ideal in the number field nf, 0 otherwise.

nfisincl (*x*, *y*)
Tests whether the number field K defined by the polynomial x is conjugate to a subfield of the field L defined by y (where x and y must be in $\mathbb{Q}[X]$). If they are not, the output is the number 0. If they are, the output is a vector of polynomials, each polynomial a representing an embedding of K into L, i.e. being such that $y||xoa$.

If y is a number field (*nf*), a much faster algorithm is used (factoring x over y using `nffactor`). Before version 2.0.14, this wasn't guaranteed to return all the embeddings, hence was triggered by a special flag. This is no more the case.

nfisisom (*x*, *y*)
As `nfisincl` , but tests for isomorphism. If either x or y is a number field, a much faster algorithm will be used.

nfkermodpr (*nf*, *x*, *pr*)
Kernel of the matrix a in $\mathbb{Z}_K/pr$, where *pr* is in **modpr** format (see `nfmodprinit`).

nfmodprinit (*nf*, *pr*)
Transforms the prime ideal *pr* into `modpr` format necessary for all operations modulo *pr* in the number field *nf*.

nfnewprec (*nf*, *precision=0*)
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) but may be quite slow: many generators of principal ideals have to be computed; note that in this latter case, the *bnf* must contain fundamental units.

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 be computed internally. This is useful in two situations: when you don't need the `nf` , or when you can't compute its discriminant due to integer factorization difficulties. In the latter case, `addprimes` is a possibility but a dangerous one: roots will probably be missed if the (true) field discriminant and an `addprimes` entry are strictly divisible by some prime. If you have such an unsafe *nf*, it is safer to input `nf.pol` .

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.

```
? 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=0*)

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 $i >= 2$. If $flag$ is non-zero 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 3-component 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

$$(b_1e_1 \oplus ... \oplus b_ne_n)/(a_1A_1 \oplus ... \oplus a_nA_n),$$

The integrality condition is $a_{i,j} \in b_ia_j^{-1}$ 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*)

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 $a.x = b$; 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 (*nf*, *d=None*)

Defining polynomial over $\mathbb{Q}$ for the splitting field of *nf*; if d is given, it must be a multiple of the splitting field degree. Instead of `nf`, it is possible to input a defining (irreducible) polynomial T for `nf`, but in general this is less efficient.

```
? K = nfinit(x^3-2);
? nfsplitting(K)
%2 = x^6 + 108
? nfsplitting(x^8-2)
%3 = x^16 + 272*x^8 + 64
```

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=0*)

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 $[g, h]$, 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 J. Klüners's algorithm in the general case, and B. Allombert's `galoissubfields` when *nf* is Galois (with weakly supersolvable Galois group).

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 `norml2`). 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=None*, *precision=0*)

L^p-norm of x; sup norm if p is omitted. 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, `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
? M = [1,-2;-3,4]; normlp(M) \\ matrix
%2 = 4
? T = (1+I) + I*x^2; normlp(T)
%3 = 1.4142135623730950488016887242096980786
? normlp([[1,2], [3,4], 5, 6]) \\ recursively defined
%4 = 6

? normlp(v, 1)
%5 = 6
? normlp(M, 1)
%6 = 10
? normlp(T, 1)
%7 = 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 (*x*)

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

```
numerator( content(x) )
```

instead.

In other cases, `numerator(x)` is defined to be `denominator(x)*x` . This is the case when x is a vector or a matrix, but also for `t_COMPLEX` or `t_QUAD` . In particular since a `t_PADIC` or `t_INTMOD` has denominator 1, its numerator is itself.

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).

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
```

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 extension of $\mathbb{Q}_p$, given as a `t_POLMOD` at least one of whose coefficients is a `t_PADIC` . In this case, the result is the vector of roots belonging to the same extension of $\mathbb{Q}_p$ as a.

padicfields (*p*, *N*, *flag=0*)
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 non-isomorphic extensions; in particular, the number of polynomials is the number of classes of non-isomorphic 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 `+oo` if x is an exact object (as a p-adic):

```
? padicprec((1 + O(2^5)) * x + (2 + O(2^4)), 2)
%1 = 4
? padicprec(x + 2, 2)
%2 = +oo
? padicprec(2 + x + O(x^2), 2)
%3 = +oo
```

The function raises an exception if it encounters an object incompatible with p-adic computations:

```
? padicprec(O(3), 2)
 *** at top-level: padicprec(O(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 (*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.

parselect (*f*, *A*, *flag=0*)

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.

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.

polclass (*D*, *inv=0*, *x=None*)

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 non-inert, 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 $P^{\pm 1}Q^{\pm 1}$ 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 : $\gamma_2 = j^{1/3}$, $D = 1, 2 mod 3$;

- 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 : $w_{3,3}$, $D = 1 mod 2$;
- 10: $w_{2,5}$, $D! = 60 mod 80$ and $D = 1, 2 mod 3$;
- 14: $w_{2,7}$, $D = 1 mod 8$;
- 15: $w_{3,5}$, $D = 1, 2 mod 3$;
- 21: $w_{3,7}$, $D = 1 mod 2$ and $21 \nmid D$
- 23: $w_{2,3}^2$, $D = 1, 2 mod 3$;
- 24: $w_{2,5}^2$, $D = 1, 2 mod 3$;
- 26: $w_{2,13}$, $D! = 156 mod 208$;
- 27: $w_{2,7}^2$, $D! = 28 mod 112$;
- 28: $w_{3,3}^2$, $D = 1, 2 mod 3$;
- 35: $w_{5,7}$, $D = 1, 2 mod 3$;
- 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
```

polcoeff (*x*, *n*, *v=None*)

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.

For greater flexibility, x can be a vector or matrix type and the function then returns `component(x,n)`.

polcompositum (*P*, *Q*, *flag=0*)

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 + 5*x^19 + 15*x^18 + 35*x^17 + 70*x^16 + 141*x^15 + 260*x^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 \\ 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) \\ 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]
```

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 =
[ x^2 + x + 1 1]

[x^4 + x^3 + x^2 + x + 1 1]
```

poldegree (*x*, *v=None*)

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 non-zero scalar is 0. Finally, when x is a non-zero polynomial or rational function, returns the ordinary degree of x. Raise an error otherwise.

poldisc (*pol*, *v=None*)

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
```

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=0*)

Galois group of the non-constant polynomial $T \in \mathbb{Q}[X]$. In the present version **2.8.0**, 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 = t^3 - 2; p = 7; A = x^2 + t + 1;
 B = [x + (3*t^2 + t + 1), x + (4*t^2 + 6*t + 6)];
 r = polhensellift(A, B, [p, T], 6) }
%1 = [x + (20191*t^2 + 50604*t + 75783), x + (97458*t^2 + 67045*t + 41866)]
? liftall( r[1] * r[2] * Mod(Mod(1,p^6),T) )
%2 = x^2 + (t + 1)
```

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 **2.8.0**), returns 1 if *pol* is non-constant and irreducible, 0 otherwise. Irreducibility is checked over the smallest base field over which *pol* seems to be defined.

pollead (*x*, *v=None*)

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 (*pol*)

Reciprocal polynomial of *pol*, i.e. the coefficients are in reverse order. *pol* must be a polynomial.

polred (*T*, *flag=0*, *_arg2=None*)

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 is equal to $x - 1$), 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 x^2 + 2]

[1/4*x^3 x^4 + 2]

[x x^4 + 8]
? minpoly(Mod(M[2,1], x^4+8))
%2 = x^2 + 2
```

polredabs (*T*, *flag=0*)

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. non-monic polynomials, or pairs `[T,listP]` specifying that a non-maximal order may be used.

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 $\mathrm{disc}(T)$ (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 use `polredbest` or `polredabs(,16)` to avoid this factorization step; in both cases, the result is 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 $\pm P(-x)$, only one is given.

16: Possibly use a suborder of the maximal order, *without* attempting to certify the result as in Warning 1: we always return a polynomial and never 0. The result is a priori not canonical.

```
? 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 = polredabs(T); T2 = polredbest(T);
? [ norml2(polroots(T1)), norml2(polroots(T2)) ]
%3 = [88.0000000, 120.000000]
```

```
? [ sizedigit(poldisc(T1)), sizedigit(poldisc(T2)) ]
%4 = [75, 67]
```

polredbest (*T*, *flag=0*)

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. non-monic 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 the ring of integers of $\mathbb{Q}[X]/(T)$, then examines small linear combinations of the basis vectors, computing their characteristic polynomials. It returns the *separable* P polynomial 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(x^4 + 8, 1)
%1 = [x^4 + 2, Mod(x^3, x^4 + 2)]
? charpoly(a)
%2 = x^4 + 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*)

Finds polynomials with reasonably small coefficients and of the same degree as that of x defining suborders of the order defined by x. One of the polynomials always defines $\mathbb{Q}$ (hence is equal to $(x-1)^n$, where n is the degree), and another always defines the same order as x if x is irreducible. Useless function: try `polredbest`.

polresultant (*x*, *y*, *v=None*, *flag=0*)

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.

polresultantext (*A*, *B*, *v=None*)

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 (*x*, *precision=0*)

Complex roots of the polynomial *x*, given as a column vector where each root is repeated according to its multiplicity. The precision is given as for transcendental functions: in GP it is kept in the variable `realprecision` and is transparent to the user, but it must be explicitly given as a second argument in library mode.

The algorithm used is a modification of A. Schönhage's root-finding algorithm, due to and originally implemented by X. Gourdon. Barring bugs, it is guaranteed to converge and to give the roots to the required accuracy.

polrootsff (*x*, *p=None*, *a=None*)

Returns the vector of distinct roots of the polynomial x in the field $\mathbb{F}_q$ defined by the irreducible polynomial a over $\mathbb{F}_p$. The coefficients of x must be operation-compatible with $\mathbb{Z}/p\mathbb{Z}$. Either a or p can omitted (in which case both are ignored) if x has `t_FFELT` coefficients:

```
? polrootsff(x^2 + 1, 5, y^2+3) \\ over F_5[y]/(y^2+3) ~ F_25
%1 = [Mod(Mod(3, 5), Mod(1, 5)*y^2 + Mod(3, 5)),
 Mod(Mod(2, 5), Mod(1, 5)*y^2 + Mod(3, 5))]
? t = ffgen(y^2 + Mod(3,5), 't); \\ a generator for F_25 as a t_FFELT
? polrootsff(x^2 + 1) \\ not enough information to determine the base field
 *** at top-level: polrootsff(x^2+1)
 *** ^-----------------
 *** polrootsff: incorrect type in factorff.
? polrootsff(x^2 + t^0) \\ make sure one coeff. is a t_FFELT
%3 = [3, 2]
? polrootsff(x^2 + t + 1)
%4 = [2*t + 1, 3*t + 4]
```

Notice that the second syntax is easier to use and much more readable.

polrootsmod (*pol*, *p*, *flag=0*)

Row vector of roots modulo p of the polynomial *pol*. Multiple roots are *not* repeated.

```
? polrootsmod(x^2-1,2)
%1 = [Mod(1, 2)]~
```

If p is very small, you may set $flag = 1$, which uses a naive search.

polrootspadic (*x*, *p*, *r*)

Vector of p-adic roots of the polynomial *pol*, given to p-adic precision r 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, 5)
%1 = [2^3 + O(2^5), 2^3 + 2^4 + O(2^5), 1 + O(2^5)]~
```

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 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 eact rational coefficients.

polrootsreal (*T*, *ab=None*, *precision=0*)

Real roots of the polynomial T with rational coefficients, multiple roots being included according to their multiplicity. 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])
%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 $h >= 2$ and *ab* is omitted, the routine will apply the algorithm to Q (restricting to non-negative 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=None*, *_arg2=None*)

Number of real roots of the real squarefree 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, we use Sturm's algorithm and the result may be wrong due to round-off errors.

```
? 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
 *** at top-level: polsturm(T^2,[0,4])
 *** ^-------------------
 *** polsturm: domain error in polsturm: issquarefree(pol) = 0
? polsturm((T*1.)^2, [0, 4]) \\ not squarefree AND inexact
 *** at top-level: polsturm((T*1.)^2,[0
 *** ^--------------------
 *** polsturm: precision too low in polsturm.
```

In the last example, the input polynomial is not squarefree but there is no way to ascertain it from the given floating point approximation: we get a precision error in this case.

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.

poltschirnhaus (*x*)
Applies a random Tschirnhausen transformation to the polynomial x, which is assumed to be non-constant 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.

powers (*x*, *n*, *x0=None*)
For non-negative 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=0*)
The function behaves differently according to whether n is present and positive or not. If n is missing, the function returns the precision in decimal digits of the PARI object x. If x is an exact object, the function returns `+oo`.

```
? precision(exp(1e-100))
%1 = 154 \\ 154 significant decimal digits
? precision(2 + x)
%2 = +oo \\ exact object
? precision(0.5 + O(x))
%3 = 38 \\ floating point accuracy, NOT series precision
? precision( [ exp(1e-100), 0.5 ] )
%4 = 38 \\ minimal accuracy among components
```

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 exact or 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 $x <= 1$. 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 .

primepi (*x*)
The prime counting function. Returns the number of primes p, $p <= x$.

```
? 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.

primes (*n*)
Creates a row vector whose components are the first n prime numbers. (Returns the empty vector for $n <= 0$.) 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]
```

psi (*x*, *precision=0*)
The ψ-function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

qfauto (*G*, *fl=None*)
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 $G = {}^tHGH$ 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=0*)
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=0*)

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 above range, assuming GRH. We currently have no counter-examples but they should exist: we'd 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:

```
QFBclassno(D) =
qfbclassno(D) * if (D < 0 || norm(quadunit(D)) < 0, 1, 2)
```

Here are a few examples:

```
? qfbclassno(400000028)
time = 3,140 ms.
%1 = 1
? quadclassunit(400000028).no
time = 20 ms. \\{ much faster}
%2 = 1
? qfbclassno(-400000028)
time = 0 ms.
%3 = 7253 \\{ correct, and fast enough}
? 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, where x is non-negative and congruent to 0 or 3 modulo 4. For $x > 5.10^5$, we assume the GRH, and use `quadclassunit` with default parameters.

qfbil (*x*, *y*, *q=None*)

Evaluate the bilinear form q (symmetric matrix) at the vectors (x, y); if q omitted, use the standard Euclidean scalar product, corresponding to the identity matrix.

Roughly equivalent to `x~ * q * y` , but a little faster and more convenient (does not distinguish between column and row vectors):

```
? x = [1,2,3]~; y = [-1,0,1]~; qfbil(x,y)
%1 = 2
? q = [1,2,3;2,2,-1;3,-1,0]; qfbil(x,y, q)
%2 = -13
? for(i=1,10^6, qfbil(x,y,q))
%3 = 568ms
? for(i=1,10^6, x~*q*y)
%4 = 717ms
```

The attached quadratic form is also available, as `qfnorm` , slightly faster:

```
? for(i=1,10^6, qfnorm(x,q))
time = 444ms
? for(i=1,10^6, qfnorm(x))
time = 176 ms.
? for(i=1,10^6, qfbil(x,y))
time = 208 ms.
```

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=None*)

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 non-negative and $n < 2^{31}$.

qfbprimeform (*x*, *p*, *precision=0*)

Prime binary quadratic form of discriminant x whose first coefficient is p, where $\|p\|$ is a prime number. By abuse of notation, $p = \pm 1$ 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=0*, *d=None*, *isd=None*, *sd=None*)

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=None*)

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 `t_QFR` , the distance component is unaffected.

qfbsolve (*Q*, *p*)

Solve the equation $Q(x, y) = p$ over the integers, where Q is a binary quadratic form and p a prime number.

Return $[x, y]$ as a two-components vector, or zero if there is no solution. Note that this function returns only one solution and not all the solutions.

Let $D = \mathrm{disc}Q$. The algorithm used runs in probabilistic polynomial time in p (through the computation of a square root of D modulo p); it is polynomial time in 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 in D assuming the GRH.

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=None*)

G, H being square and symmetric matrices with integer entries representing positive definite quadratic forms, return an invertible matrix S such that $G = {}^tSHS$. 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*.

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=None*, *m=None*)

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.
- `fl[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=0*)

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.

```
? \p19
? A = [1,2;2,1]; mateigen(A)
%1 =
[-1 1]

[ 1 1]
? [L, H] = qfjacobi(A);
? L
%3 = [-1.000000000000000000, 3.000000000000000000]~
? H
%4 =
[ 0.7071067811865475245 0.7071067811865475244]

[-0.7071067811865475244 0.7071067811865475245]
? 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=0*)

LLL algorithm applied to the *columns* of the matrix x. The columns of x may be linearly dependent. 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. 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 $\mu_{i,j} <= \|0.51\|$, 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 $\|v_i \pm v_j\| >= \|v_i\|$ 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 $flag = 4$, 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.

qflllgram (*G*, *flag=0*)

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=None*, *m=None*, *flag=0*, *precision=0*)

x being a square and symmetric matrix 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 vectors (no limit if m is omitted). The function searches for the minimal non-zero 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), returns at most $2m$ vectors. The result is a three-component vector, the first component being the number of vectors enumerated (which may be larger than $2m$), the second being the maximum norm found, and the last vector is a matrix whose columns are found vectors, only one being given for each pair $\pm v$ (at most m such pairs, unless m was omitted). The vectors are returned in no particular order.

If $flag = 1$, ignores m and returns $[N, v]$, where v is a non-zero vector of length $N <= b$, or [] if no non-zero vector has length $<= b$. If no explicit b is provided, return a vector of smallish norm (smallest vector in an LLL-reduced basis).

In these two cases, x must have *integral* entries. The implementation uses low precision floating point computations for maximal speed, which gives incorrect result when x has large entries. (The 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. In this case, if b is omitted, the "minimal" vectors only have approximately the same norm. If b is omitted, m is an upper bound for the number of vectors that will be stored and returned, but all minimal vectors are nevertheless enumerated. If m is omitted, all vectors found are stored and returned; note that this may be a huge vector!

```
? x = matid(2);
? qfminim(x) \\ 4 minimal vectors of norm 1: ±[0,1], ±[1,0]
%2 = [4, 1, [0, 1; 1, 0]]
? { x =
[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, 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, 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,-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,-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) \\ the Leech lattice has 196560 minimal vectors of norm 4
time = 648 ms.
%4 = [196560, 4, [;]]
? qfminim(x,,0,2); \\ safe algorithm. Slower and unnecessary here.
time = 18,161 ms.
%5 = [196560, 4.000061035156250000, [;]]
```

In the last example, we store 0 vectors to limit memory use. All minimal vectors are nevertheless enumerated. Provided `parisize` is about 50MB, `qfminim(x)` succeeds in 2.5 seconds.

qfnorm (*x*, *q=None*)

Evaluate the binary quadratic form q (symmetric matrix) at the vector x. If q omitted, use the standard Euclidean form, corresponding to the identity matrix.

Equivalent to `x~ * q * x`, but about twice faster and more convenient (does not distinguish between column and row vectors):

```
? x = [1,2,3]~; qfnorm(x)
%1 = 14
? q = [1,2,3;2,2,-1;3,-1,0]; qfnorm(x, q)
%2 = 23
? for(i=1,10^6, qfnorm(x,q))
time = 384ms.
? for(i=1,10^6, x~*q*x)
time = 729ms.
```

We also allow `t_MAT` s of compatible dimensions for x, and return `x~ * q * x` in this case as well:

```
? M = [1,2,3;4,5,6;7,8,9]; qfnorm(M) \\ Gram matrix
%5 =
[66 78 90]

[78 93 108]

[90 108 126]

? for(i=1,10^6, qfnorm(M,q))
time = 2,144 ms.
? for(i=1,10^6, M~*q*M)
time = 2,793 ms.
```

The polar form is also available, as `qfbil` .

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 A_1^6.

```
? Q=matid(6); G=qfauto(Q); V=qfminim(Q,3);
? apply(x->[x[1],#x],qforbits(G,V))
%2 = [[[0,0,0,0,0,1]~,6],[[0,0,0,0,1,-1]~,30],[[0,0,0,1,-1,-1]~,80]]
```

qfparam (*G*, *sol*, *flag=0*)
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,x/y,(x/y)^2]~
%3 = [3*x^2 - 10*y*x - 3*y^2, -5*x^2 - 6*y*x + 5*y^2, -x^2 - y^2]~
? v~*G*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 $v_i v_i^t$, where s is half the number of minimal vectors and the v_i ($1 <= i <= s$) are the minimal vectors.

Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension of x grows.

qfrep (*q*, *B*, *flag=0*)
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, $1 <= i <= B$ 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, $1 <= i <= B$ 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 $n >= 1$, solve over $\mathbb{Q}$ the quadratic equation $X^tGX = 0$. The matrix G must have rational coefficients. The solution might be a single non-zero 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=0*, *tech=None*, *precision=0*)

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 `quadregula` 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 non-negative 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 $<= c_1(\log \|D\|)^2$, then prove that ideals of norm $<= c_2(\log \|D\|)^2$ 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 `coredisc` (d) where d is the integer square-free 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*)
Creates the quadratic number $\omega = (a + \sqrt{D})/2$ where $a = 0$ if $D = 0mod4$, $a = 1$ if $D = 1mod4$, 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.

quadhilbert (*D*, *precision=0*)
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=None*)
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.

quadray (*D*, *f*, *precision=0*)
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=0*)
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*)
Fundamental unit 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)).

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 + O(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*)

Returns a strong pseudo prime (see `ispseudoprime`) in $[2, N-1]$. A `t_VEC` $N = [a, b]$ is also allowed, with $a <= b$ in which case a pseudo prime $a <= p <= b$ 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.

real (*x*)

Real part of x. In the case where 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 an 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.

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=None*)

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*, *pol*)

Given bnf as output by `bnfinit` , and *pol* a relative polynomial defining an Abelian extension, computes the class field theory conductor of this Abelian extension. The result is a 3-component vector $[conductor, bnr, subgroup]$, where *conductor* 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 *subgroup* is a matrix in HNF defining the subgroup of the ray class group on `bnr.gen` .

rnfdedekind (*nf*, *pol*, *pr=None*, *flag=0*)

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 non-zero, 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^2-3); P = x^3 - 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 non-monic $P = \sum_{i<=n} a_i X^i \in \mathbb{Z}_K[X]$ if $flag = 1$; in this case, we test maximality of Dedekind's order generated by

$$1, a_n\alpha, a_n\alpha^2 + a_{n-1}\alpha, ..., a_n\alpha^{n-1} + a_{n-1}\alpha^{n-2} + ... + a_1\alpha.$$

The routine will fail if P is 0 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*, *pol*)

Given a number field nf as output by `nfinit` and a polynomial *pol* with coefficients in nf defining a relative extension L of nf, computes the relative discriminant of L. This is a two-element row vector $[D, d]$, where D is the relative ideal discriminant and d is the relative discriminant considered as an element of nf^*/nf^{*2}. The main variable of nf *must* be of lower priority than that of *pol*, see `priority` (in the PARI manual).

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.pol
%2 = x^4 + 1
? rnfeltabstorel(L, Mod(x, L.pol))
%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,
 *** ^--------------------
 *** 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 (*rnf*, *x*, *flag=0*)

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]~
? 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]~
? 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^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 (*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=0*)
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=0*)
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^2+1, x^2 - y)
%1 = x^4 + 1
? T = y^3-2; rnfequation(nfinit(T), (x^3-2)/(x-Mod(y,T)))
%2 = x^6 + 108 \\ Galois closure of Q(2^(1/3))
```

If $flag$ is non-zero, 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 $\theta = \beta + k\alpha$ is a root of z, where β is a root of pol.

```
? 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 (*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 x be an ideal of the absolute extension $L/\mathbb{Q}$ given by a $\mathbb{Z}$-basis of elements of L. 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.

The reason why the input does not use the customary HNF in terms of a fixed $\mathbb{Z}$-basis for $\mathbb{Z}_L$ is precisely that no such basis has been explicitly specified. On the other hand, if you already computed an (absolute) `nf` structure `Labs` attached to L, and m is in HNF, defining an (absolute) ideal with respect to the $\mathbb{Z}$-basis `Labs.zk` , then `Labs.zk * m` is a suitable $\mathbb{Z}$-basis for the ideal, and

```
rnfidealabstorel(rnf, Labs.zk * m)
```

converts m to a relative ideal.

```
? K = nfinit(y^2+1); L = rnfinit(K, x^2-y); Labs = nfinit(L);
? m = idealhnf(Labs, 17, x^3+2);
? B = rnfidealabstorel(L, Labs.zk * m)
%3 = [[1, 8; 0, 1], [[17, 4; 0, 1], 1]] \\ pseudo-basis for m as Z_K-module
? A = rnfidealreltoabs(L, B)
%4 = [17, x^2 + 4, x + 8, x^3 + 8*x^2] \\ Z-basis for m in Q[x]/(L.pol)
? mathnf(matalgtobasis(Labs, A))
%5 =
[17 8 4 2]

[ 0 1 0 0]
```

```
[ 0 0 1 0]

[ 0 0 0 1]
? % == m
%6 = 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.

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 we return 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=0*)

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=0*)

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*, *pol*, *flag=0*)

nf being a number field in `nfinit` format considered as base field, and *pol* a polynomial defining a relative extension over nf, this computes data to work in the relative extension. The main variable of *pol* must be of higher priority (see `priority` (in the PARI manual)) than that of nf, and the coefficients of *pol* must be in nf.

The result is a row vector, whose components are technical. In the following description, we let K be the base field defined by nf and L/K the extension attached to the *rnf*. Furthermore, 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)` .

$rnf[1]$‘(: $literal$: ‘$rnf.pol$) contains the relative polynomial *pol*.

$rnf[2]$ 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` .

$rnf[4]$‘(: $literal$: ‘$rnf.index$) is the ideal index f, i.e. such that $d(pol)\mathbb{Z}_K = f^2 d(L/K)$.

$rnf[5]$ is currently unused.

$rnf[6]$ is currently unused.

$rnf[7]$ (`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 *pol*) 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,pol,1)` . Namely, a vector $[P, a, k, K.pol, pol]$ 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 *pol* and α the generator of K given above, then $P(\beta + k\alpha) = 0$.

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, pol, 1);
if (k, pol = subst(pol, x, x - k*Mod(y, K.pol)));
L = rnfinit(K, pol);
```

$rnf[12]$ 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).

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^2 + 23);
? rnfisabelian(K, x^3 - 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.

rnfisnorm (*T*, *a*, *flag=0*)

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 $a = \mathrm{Norm}(x) * q$. 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; 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 $12\log^2\|\mathrm{disc}(M)\|$, 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^3 + y^2 - 2*y - 1);
p = x^2 + Mod(y^2 + 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=2*)

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=None*, *d=0*, *precision=0*)

bnr being as output by `bnrinit`, finds a relative equation for the class field corresponding to the module in *bnr* and the given congruence subgroup (the full ray class field if *subgp* is omitted). If d is positive, outputs the list of all relative equations of degree d contained in the ray class field defined by *bnr*, with the *same* conductor as $(bnr, subgp)$.

Warning. This routine only works for subgroups of prime index. It uses Kummer theory, adjoining necessary roots of unity (it needs to compute a tough `bnfinit` here), and finds a generator via Hecke's characterization of ramification in Kummer extensions of prime degree. If your extension does not have prime degree, for the time being, you have to split it by hand as a tower / compositum of such extensions.

rnflllgram (*nf*, *pol*, *order*, *precision=0*)
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=0*)
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 **2.8.0**, 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=0*)
THIS FUNCTION IS OBSOLETE: use `rnfpolredbest` instead. Relative version of `polredabs` . Given a monic polynomial *pol* with coefficients in nf, finds a simpler relative polynomial defining the same field. The binary digits of $flag$ mean

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. Same as but faster than

```
rnfequation(nf, rnfpolredabs(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: possibly use a suborder of the maximal order. This is slower than the default when the relative discriminant is smooth, and much faster otherwise. See `polredabs` (in the PARI manual).

Warning. In the present implementation, `rnfpolredabs` produces smaller polynomials than `rnfpolred` and is usually faster, but its complexity is still exponential in the absolute degree. The function `rnfpolredbest` runs in polynomial time, and tends to return polynomials with smaller discriminants.

rnfpolredbest (*nf*, *pol*, *flag=0*)
Relative version of `polredbest` . Given a monic 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^6-3*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*, *pol*)

Given a number field nf as output by `nfinit` and a polynomial *pol* with coefficients in nf defining a relative extension L of nf, computes a pseudo-basis (A, I) for the maximal order $\mathbb{Z}_L$ viewed as a $\mathbb{Z}_K$-module, and the relative discriminant of L. This is output as a four-element row vector $[A, I, D, d]$, where D is the relative ideal discriminant and d is the relative discriminant considered as an element of nf^*/nf^{*2}.

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 $SK_0(\mathbb{Z}_K)$).

select (*f*, *A*, *flag=0*)

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 non-zero. 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]
```

returns the primes of the form $i^2 + 1$ for some $i <= 50$, 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_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]
```

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 non-0 entries in `M`. If the latter is a `t_MAT`, we extract the matrix of non-0 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 (*s*, *p*, *r*)

finds a linear relation between powers $(1, s, ..., s^p)$ of the series s, with polynomial coefficients of degree $<= r$. 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)
%2 = -x^2 + (8*y^2 + 20*y + 1)
? subst(%, x, s)
```

```
%3 = O(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.

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 non-negative 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 respec 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 + O(y^2), y)
%1 = 2
? serprec(x + 2, x)
%2 = +oo
? serprec(2 + x + O(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 + O(x^8)
? tan(t)
%3 = x + O(x^8)
```

setbinop (*f*, *X*, *Y=None*)

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 either a technical array output by `getrand` , or a small positive integer, used to generate deterministically a suitable state array. For instance, running a randomized computation starting by `setrand(1)` twice will generate the exact same output.

setsearch (*S*, *x*, *flag=0*)
Determines whether x belongs to the set S (see `setisset`).

We first describe the default behaviour, 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 non-zero, 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 $n >= 0$ 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=1*)
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=0*)
Sine of x.

sinc (*x*, *precision=0*)
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=0*)
Hyperbolic sine of x.

sizebyte (*x*)
Outputs the total number of bytes occupied by the tree representing the PARI object x.

sizedigit (*x*)
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

$$floor(1+\log_2 x)\log_{10} 2,$$

This function is DEPRECATED, essentially meaningless, and provided for backwards compatibility only. Don't use it!

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 + O(2^4))^2
%1 = 1 + O(2^5)
? (1 + O(2^4)) * (1 + O(2^4))
%2 = 1 + O(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 + O(2^4)); x * x
%3 = 1 + O(2^5)
? (1 + O(2^4))^4
%4 = 1 + O(2^6)
```

(note the difference between `%2` and `%3` above).

sqrt (*x*, *precision=0*)
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*)
Returns the integer square root of x, i.e. the largest integer y such that $y^2 <= x$, where x a non-negative integer.

```
? N = 120938191237; sqrtint(N)
%1 = 347761
? sqrt(N)
%2 = 347761.68741970412747602130964414095216
```

sqrtnint (*x*, *n*)
Returns the integer n-th root of x, i.e. the largest integer y such that $y^n <= x$, where x is a non-negative integer.

```
? N = 120938191237; sqrtnint(N, 5)
%1 = 164
? N^(1/5)
%2 = 164.63140849829660842958614676939677391
```

The special case $n = 2$ is `sqrtint`

subgrouplist (*bnr*, *bound=None*, *flag=0*)
bnr being as output by `bnrinit` or a list of cyclic components of a finite Abelian group G, 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 *bnr* is as output by `bnrinit` , gives only the subgroups whose modulus is the conductor. Otherwise, the modulus is not taken into account.

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 $120oo$ (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 `rnfkummer` .)

subst (*x*, *y*, *z*)

Replace the simple variable y by the argument z in the "polynomial" expression x. 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. Finally, 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^4 + x^2 + 1, x^2, y)
%1 = y^2 + y + 1
? substpol(x^4 + x^2 + 1, x^3, y)
%2 = x^2 + y*x + 1
? substpol(x^4 + x^2 + 1, (x+1)^2, y)
%3 = (-4*y - 6)*x + (y^2 + 3*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=None*)

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.

sumformal (*f*, *v=None*)

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) \\ 1 + ... + n
%1 = 1/2*n^2 + 1/2*n
? f(n) = n^3+n^2+1;
? F = sumformal(f(n)) \\ 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
```

sumnuminit (*asymp*, *precision=0*)

Initialize tables for Euler–MacLaurin delta summation of a series with positive terms. If given, `asymp` is of the form $[+oo, \alpha]$, as in `intnum` 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.
```

sumnummonieninit (*asymp*, *w=None*, *n0=None*, *precision=0*)

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 non-zero asymptotic expansion

$$f(n) = \sum_{i>=2} a_i/n^i$$

at infinity. Note that the sum starts at $i = 2$! The argument `asymp` allows to specify different expansions:

•a real number $\alpha > 1$ means

$$f(n) = \sum_{i>=1} a_i/n^{\alpha}$$

$$(Nowthesummationstartsat : math : `1`.)$$

•a vector $[\alpha, \beta]$ of reals, where we must have $\alpha > 0$ and $\alpha + \beta > 1$ to ensure convergence, means that

$$f(n) = \sum_{i>=1} a_i/n^{\alpha i+\beta}$$

$$Notethat : math : `asymp = [\alpha, \alpha]`isequivalentto : math : `asymp = \alpha`.$$

```
? \p38
? s = sumnum(n = 1, sin(1/sqrt(n)) / n)
%1 = 2.3979771206715998375659850036324914714

? sumnummonien(n = 1, sin(1/sqrt(n)) / n) - s
%2 = -0.001[...] \\ completely wrong !

? t = sumnummonieninit([1/2,1]); \\ f(n) = sum_i 1 / n^(i/2+1)
? sumnummonien(n = 1, sin(1/sqrt(n)) / n, t) - s
%3 = 0.E-37 \\ now correct
```

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 $m >= 1$. Note that for non-negative 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 `sumnum`. The following variants are available

- an integer $k >= 0$, to code $w(n) = (\log n)^k$; only the cases $k = 0, 1$ are presently implemented; due to a poor implementation of ζ derivatives, it is not currently worth it to exploit the special shape of g_w when $k > 0$;
- 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 $w = w$.
- a vector $[w, oo]$, where w is a closure as above; we assume that $w(n)$ decreases exponentially. Note that in this case, `sumnummonien` is provided for completeness and comparison purposes only: one of `suminf` or `sumpos` should be preferred in practice.

The cases where w is a closure or $w(n) = \log n$ are the only ones where n_0 is taken into account and stored in the result. The subsequent call to `sumnummonien` *must* use the same value.

```
? \p300
? sumnummonien(n = 1, n^-2*log(n)) + zeta'(2)
time = 536 ms.
%1 = -1.323[...]E-6 \\ completely wrong, f does not satisfy hypotheses !
? tab = sumnummonieninit(, 1); \\ codes w(n) = log(n)
time = 18,316 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 44 ms.
%3 = -5.562684646268003458 E-309 \\ now perfect

? tab = sumnummonieninit(, n->log(n)); \\ generic, about as fast
time = 18,693 ms.
? sumnummonien(n = 1, n^-2, tab) + zeta'(2)
time = 40 ms.
%5 = -5.562684646268003458 E-309 \\ identical result
```

tan (*x*, *precision=0*)
Tangent of x.

tanh (*x*, *precision=0*)
Hyperbolic tangent of x.

taylor (*x*, *t*, *serprec=-1*)
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 + O(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=None*)
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 + O(101^5))
%1 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + O(101^5)
? z^100
```

```
%2 = 1 + O(101^5)
? T = teichmuller([101, 5]);
? teichmuller(2 + O(101^5), T)
%4 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + O(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+O(p^n), T)))
time = 60 ms.
? for(i=1,10^3, vector(p-1, i, teichmuller(i+O(p^n))))
time = 1,293 ms.
? 1 + 2*(log(p)/log(2) + hammingweight(p))
%8 = 22.316[...]
```

Here the precompuation 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 + O(101^5), Tlow)
%10 = 2 + 83*101 + O(101^5) \\ no longer a root of 1

? Thigh = teichmuller([101, 10]); \\ higher accuracy
? teichmuller(2 + O(101^5), Thigh)
%12 = 2 + 83*101 + 18*101^2 + 69*101^3 + 62*101^4 + O(101^5)
```

theta (*q*, *z*, *precision=0*)

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=0*)

k-th derivative at $z = 0$ of $theta(q, z)$.

thue (*tnf*, *a*, *sol=None*)

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 $\mathrm{Norm}(x) = a$ 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,0)`. 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 $a! = \pm 1$ 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 non-monic 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.

thueinit (*P*, *flag=0*, *precision=0*)

Initializes the *tnf* corresponding to P, a non-constant 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 non-zero, 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 use `bnfisintnorm`):

```
? 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 = [[-1000000000000000, 0], [1000000000000000, 0]]
```

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], [10000000000, 0]]
```

The last 2 commands use less than 20ms.

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 non-square matrix (but not a vector), an error occurs.

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` ".

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.

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 + O(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 occuring 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]
```

vecextract (*x*, *y*, *z=None*)

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 (non-zero) 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..l]` 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‘.

vecsearch (*v*, *x*, *cmpf=None*)

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).

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
```

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=None*, *flag=0*)

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). 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 an integer which is < 0, > 0 or $= 0$ if $x < y$, $x > y$ or $x = y$ respectively. The `sign` function is very useful in this context:

```
? vecsort([3,0,2; 1,0,2]) \\ sort columns according to lex order
%1 =
[0 2 3]

[0 2 1]
? vecsort(v, (x,y)->sign(y-x)) \\ reverse sort
? vecsort(v, (x,y)->sign(abs(x)-abs(y))) \\ sort by increasing absolute value
? cmpf(x,y) = my(dx = poldisc(x), dy = poldisc(y)); sign(abs(dx) - abs(dy))
? vecsort([x^2+1, x^3-2, x^4+5*x+1], cmpf)
```

The last example used the named `cmpf` instead of an anonymous function, and sorts polynomials with respect to the absolute value of their discriminant. A more efficient approach would use precomputations to ensure a given discriminant is computed only once:

```
? DISC = vector(#v, i, abs(poldisc(v[i])));
? perm = vecsort(vector(#v,i,i), (x,y)->sign(DISC[x]-DISC[y]))
? vecextract(v, perm)
```

Similar ideas apply whenever we sort according to the values of a function which is expensive to compute.

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
```

weber (*x*, *flag=0*, *precision=0*)
One of Weber's three f functions. If $flag = 0$, returns

$$f(x) = \exp(-i\pi/24).\eta((x+1)/2)/\eta(x) such that j = (f^{24} - 16)^3/f^{24},$$

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 (*s*, *precision=0*)
For s 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.

For s 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. $p = 2$).

zetamult (*s*, *precision=0*)
For s a vector of positive integers such that $s[1] >= 2$, returns the multiple zeta value (MZV)

$$\zeta(s_1, ..., s_k) = \sum_{n_1 > ... > n_k > 0} n_1^{-s_1} ... n_k^{-s_k}.$$

```
? zetamult([2,1]) - zeta(3) \\ Euler's identity
%1 = 0.E-38
```

zncharinduce (*G*, *chi*, *N*)
Let G be attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per `G = idealstar(,q)`) 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 $(\mathbb{Z}/q\mathbb{Z})^*$ or its Conrey logarithm; see `dirichletchar` (in the PARI manual) or `??character` .

Let N be a multiple of q, return the character modulo N induced by `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 `idealstar(,N)` .

```
? G = idealstar(,4);
? 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 = idealstar(,80);
? zncharinduce(G, 1, G2) \\ same
%4 = [0, 0, 0]~

? chi = zncharinduce(G, 3, G2) \\ induce the non-trivial character mod 4
%5 = [1, 0, 0]~
? znconreyconductor(G2, chi, &chi0)
%6 = [4, Mat([2, 2])]
? chi0
%7 = [1]~
```

Here is a larger example:

```
? G = idealstar(,126000);
? label = 1009;
? chi = znconreylog(G, label)
%3 = [0, 0, 0, 14, 0]~
? N0 = znconreyconductor(G, label, &chi0)
%4 = [125, Mat([5, 3])]
? chi0 \\ primitive character mod 5^3 attached to chi
%5 = [14]~
? G0 = idealstar(,N0);
? zncharinduce(G0, chi0, G) \\ induce back
%7 = [0, 0, 0, 14, 0]~
? znconreyexp(G, %)
%8 = 1009
```

zncharisodd (*G*, *chi*)

Let G be attached to $(\mathbb{Z}/N\mathbb{Z})^*$ (as per `G = idealstar(,N)`) 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 $(\mathbb{Z}/q\mathbb{Z})^*$ 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 = idealstar(,8);
? zncharisodd(G, 1) \\ trivial character
%2 = 0
? zncharisodd(G, 3)
%3 = 1
? chareval(G, 3, -1)
%4 = 1/2
```

znconreychar (*bid*, *m*)

Given a *bid* attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per `bid = idealstar(,q)`), 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 integer whose order is $\phi(p^{e_p})$.

For $p = 2$, we let g_4 (if $2^{e_2} >= 4$) and g_8 (if furthermore ($2^{e_2} >= 8$) be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to 1 mod $q/2^{e_2}$,
- $g_4 = -1 mod 2^{e_2}$,
- $g_8 = 5 mod 2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} >= 2$) 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` or `idealstar` 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 $(m_p)_{p \in S_q}$, 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 `bid.gen` .

Note. It is useless to include the generators in the *bid*, except for debugging purposes: they are well defined from elementary matrix operations and Chinese remaindering, their explicit value as elements in $(\mathbb{Z}/q\mathbb{Z})^*$ is never used.

```
? G = idealstar(,8,2); /*add generators for debugging:*/
? G.cyc
%2 = [2, 2] \\ Z/2 x Z/2
? G.gen
%3 = [7, 3]
? znconreychar(G,1) \\ 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 = idealstar(,55);
? znconreychar(G,7)
%2 = [7, 0]
? znconreychar(G, znconreylog(G,7))
%3 = [7, 0]
```

znconreyexp (*bid*, *chi*)

Given a *bid* attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per `bid = idealstar(,q)`), this function returns the Conrey exponential of the character *chi*: it returns the integer $m \in (\mathbb{Z}/q\mathbb{Z})^*$ such that `znconreylog(:emphasis:`bid`, m)‘ is *chi*.

The character *chi* is given either as a

- `t_VEC` : in terms of the generators `:emphasis:`bid`.gen‘;
- `t_COL` : a Conrey logarithm.

```
? G = idealstar(,126000)
? 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]~
? 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 (*bid*, *m*)

Given a *bid* attached to $(\mathbb{Z}/q\mathbb{Z})^*$ (as per `bid = idealstar(,q)`), 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 $e_2 >= 2$. (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 integer whose order is $\phi(p^{e_p})$ for p odd,

For $p = 2$, we let g_4 (if $2^{e_2} >= 4$) and g_8 (if furthermore ($2^{e_2} >= 8$) be the elements in $(\mathbb{Z}/q\mathbb{Z})^*$ which are

- congruent to 1 mod $q/2^{e_2}$,
- $g_4 = -1 mod 2^{e_2}$,
- $g_8 = 5 mod 2^{e_2}$.

Then the g_p (and the extra g_4 and g_8 if $2^{e_2} >= 2$) 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` or `idealstar` 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 = idealstar(,126000);
? znconreylog(G,1)
%2 = [0, 0, 0, 0, 0]~
? 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]~
? 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 (*P*, *N*, *X*, *B=None*)

N being an integer and $P \in \mathbb{Z}[X]$, finds all integers x with $\|x\| <= X$ such that

$$\gcd(N, P(x)) >= B,$$

using Coppersmith's algorithm (a famous application of the LLL algorithm). X must be smaller than $\exp(\log^2 B/(\deg(P)\log N))$: for $B = N$, this means $X < N^{1/\deg(P)}$. Some x larger than X may be returned if you are very lucky. The smaller B (or the larger X), the slower the routine will be. 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 non-trivial factors of N, given some partial information on the factors; in that case B must obviously be smaller than the largest non-trivial divisor of N.

```
setrand(1); \\ to make the example reproducible
interval = [10^30, 10^31];
p = randomprime(interval);
q = randomprime(interval); N = p*q;
p0 = p % 10^20; \\ assume we know 1) p > 10^29, 2) the last 19 digits of p
L = zncoppersmith(10^19*x + p0, N, 10^12, 10^29)

\\ result in 10ms.
%6 = [738281386540]
? gcd(L[1] * 10^19 + p0, N) == p
%7 = 1
```

and we recovered p, faster than by trying all possibilities $< 10^{12}$.

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=None*)

Discrete logarithm of x in $(\mathbb{Z}/N\mathbb{Z})^*$ in base g. The result is [] when x is not a power of g. 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 = 205 ms.
? for(i=1,10^4, znlog(i, g))
time = 244 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 $(\mathbb{Z}/N\mathbb{Z})^*$ when N is prime: a linear sieve index calculus method, suitable for $N < 10^{50}$, 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)) \\ no solution
%8 = []
```

For convenience, g is also allowed to be a p-adic number:

```
? g = 3+O(5^10); znlog(2, g)
%1 = 1015243
? g^%
%2 = 2 + O(5^10)
```

znorder (*x*, *o=None*)

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 non-zero 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 $k >= 0$). 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*)

Gives the structure of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$ as a 3-component row vector v, where $v[1] = \phi(n)$ is the order of that group, $v[2]$ is a k-component row-vector d of integers $d[i]$ such that $d[i] > 1$ and $d[i] \| d[i-1]$ for $i >= 2$ and $(\mathbb{Z}/n\mathbb{Z})^*$ $\prod_{i=1}^k (\mathbb{Z}/d[i]\mathbb{Z})$, and $v[3]$ is a k-component row vector giving generators of the image of the cyclic groups $\mathbb{Z}/d[i]\mathbb{Z}$.

```
? 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]
```

According to the above definitions, `znstar(0)` is `[2,[2],[-1]]` , corresponding to $\mathbb{Z}^*$.

`sage.libs.pari.gen.`**gentoobj** (*z*, *locals={}*)

Convert a PARI gen to a Sage/Python object.

See the `python` method of `gen` for documentation and examples.

sage.libs.pari.gen. **objtogen** (*s*)

Convert any Sage/Python object to a PARI gen.

For Sage 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:

```
sage: pari([2,3,5])
[2, 3, 5]
sage: pari(Matrix(2,2,range(4)))
[0, 1; 2, 3]
sage: pari(x^2-3)
x^2 - 3
```

```
sage: a = pari(1); a, a.type()
(1, 't_INT')
sage: a = pari(1/2); a, a.type()
(1/2, 't_FRAC')
sage: a = pari(1/2); a, a.type()
(1/2, 't_FRAC')
```

Conversion from reals uses the real's own precision:

```
sage: a = pari(1.2); a, a.type(), a.precision()
(1.20000000000000, 't_REAL', 4) # 32-bit
(1.20000000000000, 't_REAL', 3) # 64-bit
```

Conversion from strings uses the current PARI real precision. By default, this is 64 bits:

```
sage: a = pari('1.2'); a, a.type(), a.precision()
(1.20000000000000, 't_REAL', 4)  # 32-bit
(1.20000000000000, 't_REAL', 3)  # 64-bit
```

But we can change this precision:

```
sage: pari.set_real_precision(35)  # precision in decimal digits
15
sage: a = pari('1.2'); a, a.type(), a.precision()
(1.2000000000000000000000000000000000, 't_REAL', 6)  # 32-bit
(1.2000000000000000000000000000000000, 't_REAL', 4)  # 64-bit
```

Set the precision to 15 digits for the remaining tests:

```
sage: pari.set_real_precision(15)
35
```

Conversion from matrices and vectors is supported:

```
sage: a = pari(matrix(2,3,[1,2,3,4,5,6])); a, a.type()
([1, 2, 3; 4, 5, 6], 't_MAT')
sage: v = vector([1.2, 3.4, 5.6])
sage: pari(v)
[1.20000000000000, 3.40000000000000, 5.60000000000000]
```

Some more exotic examples:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: pari(K)
[y^3 - 2, [1, 1], -108, 1, [[1, 1.25992104989487, 1.58740105196820; 1, -0.
↪629960524947437 + 1.09112363597172*I, -0.793700525984100 - 1.37472963699860*I],␣
↪[1, 1.25992104989487, 1.58740105196820; 1, 0.461163111024285, -2.16843016298270;
↪ 1, -1.72108416091916, 0.581029111014503], [1, 1, 2; 1, 0, -2; 1, -2, 1], [3,␣
↪0, 0; 0, 0, 6; 0, 6, 0], [6, 0, 0; 0, 6, 0; 0, 0, 3], [2, 0, 0; 0, 0, 1; 0, 1,␣
↪0], [2, [0, 0, 2; 1, 0, 0; 0, 1, 0]], []], [1.25992104989487, -0.
↪629960524947437 + 1.09112363597172*I], [1, y, y^2], [1, 0, 0; 0, 1, 0; 0, 0,␣
↪1], [1, 0, 0, 0, 0, 2, 0, 2, 0; 0, 1, 0, 1, 0, 0, 0, 0, 2; 0, 0, 1, 0, 1, 0, 1,␣
↪0, 0]]

sage: E = EllipticCurve('37a1')
sage: pari(E)
[0, 0, 1, -1, 0, 0, -2, 1, -1, 48, -216, 37, 110592/37, Vecsmall([1]),␣
↪[Vecsmall([64, 1])], [0, 0, 0, 0, 0, 0, 0, 0]]
```

Conversion from basic Python types:

```
sage: pari(int(-5))
-5
sage: pari(long(2**150))
1427247692705959881058285969449495136382746624
sage: pari(float(pi))
3.14159265358979
sage: pari(complex(exp(pi*I/4)))
0.707106781186548 + 0.707106781186548*I
sage: pari(False)
0
sage: pari(True)
1
```

Some commands are just executed without returning a value:

```
sage: pari("dummy = 0; kill(dummy)")
sage: type(pari("dummy = 0; kill(dummy)"))
<type 'NoneType'>
```

TESTS:

```
sage: pari(None)
Traceback (most recent call last):
...
ValueError: Cannot convert None to pari
```

CHAPTER TWENTYEIGHT

PARI C-LIBRARY INTERFACE

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 (trac ticket #9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (trac ticket #11611, trac ticket #11854, trac ticket #11952)
- Peter Bruin (2013-11-17): split off this file from gen.pyx (trac ticket #15185)
- Jeroen Demeyer (2014-02-09): upgrade to PARI 2.7 (trac ticket #15767)
- Jeroen Demeyer (2014-09-19): upgrade to PARI 2.8 (trac ticket #16997)
- Jeroen Demeyer (2015-03-17): automatically generate methods from `pari.desc` (trac ticket #17631 and trac ticket #17860)

EXAMPLES:

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

Arithmetic operations cause all arguments to be converted to PARI:

```
sage: type(pari(1) + 1)
<type 'sage.libs.pari.gen.gen'>
sage: type(1 + pari(1))
<type 'sage.libs.pari.gen.gen'>
```

GUIDE TO REAL PRECISION AND THE PARI LIBRARY

The default real precision in communicating with the PARI library is the same as the default Sage real precision, which is 53 bits. Inexact Pari objects are therefore printed by default to 15 decimal digits (even if they are actually more precise).

Default precision example (53 bits, 15 significant decimals):

```
sage: a = pari(1.23); a
1.23000000000000
sage: a.sin()
0.942488801931698
```

Example with custom precision of 200 bits (60 significant decimals):

```
sage: R = RealField(200)
sage: a = pari(R(1.23)); a   # only 15 significant digits printed
1.23000000000000
sage: R(a)         # but the number is known to precision of 200 bits
1.2300000000000000000000000000000000000000000000000000000000
sage: a.sin()      # only 15 significant digits printed
0.942488801931698
sage: R(a.sin())   # but the number is known to precision of 200 bits
0.94248880193169751002382356538924454146128740562765030213504
```

It is possible to change the number of printed decimals:

```
sage: R = RealField(200)    # 200 bits of precision in computations
sage: old_prec = pari.set_real_precision(60)  # 60 decimals printed
sage: a = pari(R(1.23)); a
1.23000000000000000000000000000000000000000000000000000000000
sage: a.sin()
0.942488801931697510023823565389244541461287405627650302135038
sage: pari.set_real_precision(old_prec)  # restore the default printing behavior
60
```

Unless otherwise indicated in the docstring, most Pari functions that return inexact objects use the precision of their arguments to decide the precision of the computation. However, if some of these arguments happen to be exact numbers (integers, rationals, etc.), an optional parameter indicates the precision (in bits) to which these arguments should be converted before the computation. If this precision parameter is missing, the default precision of 53 bits is used. The following first converts 2 into a real with 53-bit precision:

```
sage: R = RealField()
sage: R(pari(2).sin())
0.909297426825682
```

We can ask for a better precision using the optional parameter:

```
sage: R = RealField(150)
sage: R(pari(2).sin(precision=150))
0.90929742682568169539601986591174484270225497
```

Warning regarding conversions Sage - Pari - Sage: Some care must be taken when juggling inexact types back and forth between Sage and Pari. In theory, calling p=pari(s) creates a Pari object p with the same precision as s; in practice, the Pari library's precision is word-based, so it will go up to the next word. For example, a default 53-bit Sage real s will be bumped up to 64 bits by adding bogus 11 bits. The function p.python() returns a Sage object with exactly the same precision as the Pari object p. So pari(s).python() is definitely not equal to s, since it has 64 bits of precision, including the bogus 11 bits. The correct way of avoiding this is to convert pari(s).python() back into a domain with the right precision. This has to be done by the user (or by Sage functions that use Pari library functions in gen.pyx). For instance, if we want to use the Pari library to compute sqrt(pi) with a precision of 100 bits:

```
sage: R = RealField(100)
sage: s = R(pi); s
3.1415926535897932384626433833
sage: p = pari(s).sqrt()
sage: x = p.python(); x  # wow, more digits than I expected!
1.7724538509055160272981674833410973484
sage: x.prec()            # has precision 'improved' from 100 to 128?
128
sage: x == RealField(128)(pi).sqrt()  # sadly, no!
False
sage: R(x)                # x should be brought back to precision 100
1.7724538509055160272981674833
sage: R(x) == s.sqrt()
True
```

Elliptic curves and precision: If you are working with elliptic curves, you should set the precision for each method:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: eta1 = e.elleta(precision=100)[0]
sage: eta1.sage()
3.6054636014326520859158205642077267748
sage: eta1 = e.elleta(precision=180)[0]
sage: eta1.sage()
3.60546360143265208591582056420772677481026899659802474544
```

Number fields and precision: TODO

TESTS:

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

```
sage: pari('print("test")')
test
```

Check that `default()` works properly:

```
sage: pari.default("debug")
0
sage: pari.default("debug", 3)
sage: pari(2^67+1).factor()
IFAC: cracking composite
        49191317529892137643
IFAC: factor 6713103182899
        is prime
IFAC: factor 7327657
        is prime
IFAC: prime 7327657
        appears with exponent = 1
IFAC: prime 6713103182899
        appears with exponent = 1
IFAC: found 2 large prime (power) factors.
[3, 1; 7327657, 1; 6713103182899, 1]
sage: pari.default("debug", 0)
sage: pari(2^67+1).factor()
[3, 1; 7327657, 1; 6713103182899, 1]
```

class `sage.libs.pari.pari_instance.`**`PariInstance`**

Bases: *`sage.libs.pari.pari_instance.PariInstance_auto`*

Initialize the PARI system.

INPUT:

- `size` – long, the number of bytes for the initial PARI stack (see note below)
- `maxprime` – unsigned long, upper limit on a precomputed prime number table (default: 500000)

Note: In Sage, the PARI stack is different than in GP or the PARI C library. In Sage, instead of the PARI stack holding the results of all computations, it *only* holds the results of an individual computation. Each time a new Python/PARI object is computed, it it copied to its own space in the Python heap, and the memory it occupied on the PARI stack is freed. Thus it is not necessary to make the stack very large. Also, unlike in PARI, if the stack does overflow, in most cases the PARI stack is automatically increased and the relevant step of the computation rerun.

This design obviously involves some performance penalties over the way PARI works, but it scales much better and is far more robust for large projects.

Note: If you do not want prime numbers, put `maxprime=2`, but be careful because many PARI functions require this table. If you get the error message "not enough precomputed primes", increase this parameter.

List (*x=None*)

Create an empty list or convert x to a list.

EXAMPLES:

```
sage: pari.List(range(5))
List([0, 1, 2, 3, 4])
sage: L = pari.List()
sage: L
List([])
sage: L.listput(42, 1)
42
sage: L
List([42])
sage: L.listinsert(24, 1)
24
sage: L
List([24, 42])
```

PARI_ONE

PARI_TWO

PARI_ZERO

allocatemem (*s=0, sizemax=0, silent=False*)

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 Sage's PARI interface 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 doubled 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:

```
sage: pari.allocatemem(10^7)
PARI stack size set to 10000000 bytes, maximum size set to 67108864
sage: pari.allocatemem()  # Double the current size
PARI stack size set to 20000000 bytes, maximum size set to 67108864
sage: pari.stacksize()
20000000
sage: pari.allocatemem(10^6)
PARI stack size set to 1000000 bytes, maximum size set to 67108864
```

The following computation will automatically increase the PARI stack size:

```
sage: 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`:

```
sage: pari.stacksize()
16000000
```

Setting a small maximum size makes this fail:

```
sage: pari.allocatemem(10^6, 2^22)
PARI stack size set to 1000000 bytes, maximum size set to 4194304
sage: a = pari('2^100000000')
Traceback (most recent call last):
...
PariError: _^s: the PARI stack overflows (current size: 1000000; maximum size:
↪ 4194304)
You can use 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:

```
sage: pari.allocatemem(1, 2^26)
PARI stack size set to 1024 bytes, maximum size set to 67108864
sage: a = pari(2)^100000000
sage: pari.stacksize()
16777216
```

We do not allow `sizemax` less than `s`:

```
sage: pari.allocatemem(10^7, 10^6)
Traceback (most recent call last):
...
ValueError: the maximum size (10000000) should be at least the stack size␣
↪(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).

EXAMPLE:

```
sage: pari.debugstack()  # random
top =  0x60b2c60
avma = 0x5875c38
bot =  0x57295e0
size = 1000000
```

double_to_gen (*x*)

euler (*precision=0*)
Euler's constant $\gamma = 0.57721...$. Note that `Euler` is one of the few reserved names which cannot be used for user variables.

factorial (*n*)
Return the factorial of the integer n as a PARI gen.

EXAMPLES:

```
sage: pari.factorial(0)
1
sage: pari.factorial(1)
1
sage: pari.factorial(5)
120
sage: pari.factorial(25)
15511210043330985984000000
```

genus2red (*P*, *P0=None*)
Let P be a polynomial with integer coefficients. Determines the reduction of the (proper, smooth) genus 2 curve $C/\mathbf{Q}$, 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 $y^2 + Q(x)y = P(x)$.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: pari.genus2red([-5*x^5, x^3 - 2*x^2 - 2*x + 1])
[1416875, [2, -1; 5, 4; 2267, 1], x^6 - 240*x^4 - 2550*x^3 - 11400*x^2 -␣
↪24100*x - 19855, [[2, [2, [Mod(1, 2)]], []], [5, [1, []], ["[V] page 156",␣
↪[3]]], [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 Sage. It has *no* effect on the precision of computations within the pari library.

EXAMPLES:

```
sage: pari.get_real_precision()
15
```

get_series_precision ( )

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

EXAMPLES:

```
sage: pari.init_primes(200000)
```

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

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

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

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.

nth_prime (**args, **kwds*)
Deprecated: Use `prime()` instead. See trac ticket #20216 for details.

one ( )
EXAMPLES:

```
sage: pari.one()
1
```

pari_version ( )

pi (*precision=0*)
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=None*)
Chebyshev polynomial of the first kind of degree n, in the variable v.

EXAMPLES:

```
sage: pari.polchebyshev(7)
64*x^7 - 112*x^5 + 56*x^3 - 7*x
sage: pari.polchebyshev(7, 'z')
64*z^7 - 112*z^5 + 56*z^3 - 7*z
sage: pari.polchebyshev(0)
1
```

polcyclo_eval (**args, **kwds*)
Deprecated: Use `polcyclo()` instead. See trac ticket #20217 for details.

polsubcyclo (*n, d, v=None*)
polsubcyclo(n, d, v=x): return the pari list of polynomial(s) defining the sub-abelian extensions of degree d of the cyclotomic field $\mathbf{Q}(\zeta_n)$, where d divides $\phi(n)$.

EXAMPLES:

```
sage: pari.polsubcyclo(8, 4)
[x^4 + 1]
sage: pari.polsubcyclo(8, 2, 'z')
[z^2 + 2, z^2 - 2, z^2 + 1]
sage: pari.polsubcyclo(8, 1)
[x - 1]
sage: pari.polsubcyclo(8, 3)
[]
```

poltchebi (**args*, ***kwds*)
Deprecated: Use `polchebyshev()` instead. See trac ticket #18203 for details.

prime_list (**args*, ***kwds*)
Deprecated: Use `primes()` instead. See trac ticket #20216 for details.

primes (*n=None*, *end=None*)
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:

```
sage: pari.primes(3)
[2, 3, 5]
sage: pari.primes(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: pari.primes(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
sage: len(pari.primes(1000))
1000
sage: pari.primes(11,29)
[11, 13, 17, 19, 23, 29]
sage: pari.primes((11,29))
[11, 13, 17, 19, 23, 29]
sage: pari.primes(end=29)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: pari.primes(10^30, 10^30 + 100)
[1000000000000000000000000000057, 1000000000000000000000000000099]
```

TESTS:

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

primes_up_to_n (*n*)

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 Sage. It has *no* effect on the precision of computations within the pari library.

Returns the previous PARI real precision.

EXAMPLES:

```
sage: pari.set_real_precision(60)
15
sage: pari('1.2')
1.20000000000000000000000000000000000000000000000000000000000
sage: pari.set_real_precision(15)
60
```

set_series_precision (*n*)

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()`

This should not be called directly; instead, use Sage's global random number seed handling in `sage.misc.randstate` and call `current_randstate().set_seed_pari()`.

EXAMPLES:

```
sage: pari.setrand(50)
sage: a = pari.getrand()
sage: pari.setrand(a)
sage: a == pari.getrand()
True
```

TESTS:

Check that invalid inputs are handled properly (trac ticket #11825):

```
sage: 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 10^6 by default. However, the stack size is automatically doubled when needed up to some maximum.

See also:

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

EXAMPLES:

```
sage: pari.stacksize()
1000000
sage: pari.allocatemem(2^18, silent=True)
sage: 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:

```
sage: pari.allocatemem(2^18, 2^26, silent=True)
sage: pari.stacksizemax()
67108864
```

vector (*n*, *entries=None*)

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

EXAMPLES:

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

zero ( )

EXAMPLES:

```
sage: pari.zero()
0
```

class sage.libs.pari.pari_instance.**PariInstance_auto**

Bases: object

Part of the *PariInstance* class containing auto-generated functions.

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

Catalan (*precision=0*)

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.

Euler (*precision=0*)

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 $\sqrt{-1}$.

Pi (*precision=0*)

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

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
```

bernfrac (*x*)

Bernoulli number B_x, where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$,..., expressed as a rational number. The argument x should be of type integer.

bernpol (*n*, *v=None*)

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 (*x*, *precision=0*)
Bernoulli number B_x, as `bernfrac`, but B_x is returned as a real number (with the current precision).

bernvec (*x*)
Creates a vector containing, as rational numbers, the Bernoulli numbers B_0, B_2,..., B_{2x}. This routine is obsolete. Use `bernfrac` instead each time you need a Bernoulli number in exact form.

Note. This routine is implemented using repeated independent calls to `bernfrac`, which is faster than the standard recursion in exact arithmetic. It is only kept for backward compatibility: it is not faster than individual calls to `bernfrac`, its output uses a lot of memory space, and coping with the index shift is awkward.

default (*key=None*, *val=None*)
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`.

ellmodulareqn (*N*, *x=None*, *y=None*)
Return a vector [`eqn`,:math:t] where `eqn` 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 currently requires the package `seadata` to be installed and is limited to $N < 500$, N prime.

Let j be the j-invariant function. The polynomial `eqn` satisfies the following functional equation, which allows to compute the values of the classical modular polynomial Φ_N of prime level N, such that $\Phi_N(j(\tau), j(N\tau)) = 0$, while being much smaller than the latter:

- for canonical type: $P(f(\tau), j(\tau)) = P(N^s/f(\tau), j(N\tau)) = 0$, where $s = 12/\gcd(12, N-1)$;
- for Atkin type: $P(f(\tau), j(\tau)) = P(f(\tau), j(N\tau)) = 0$.

In both cases, f is a suitable modular function (see below).

The following GP function returns values of the classical modular polynomial by eliminating $f(\tau)$ in the above two equations, for $N <= 31$ or $N \in 41, 47, 59, 71$.

```
classicaleqn(N, X='X, Y='Y)=
{
 my(E=ellmodulareqn(N), P=E[1], t=E[2], Q, d);
 if(poldegree(P,'y)>2,error("level unavailable in classicaleqn"));
 if (t == 0,
 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);
 ,
 Q = subst(P,'y,Y);
 d = (X-Y)^(N+1));
 polresultant(subst(P,'y,X), Q) / d;
}
```

More precisely, let $W_N(\tau) = (-1)/(N\tau)$ be the Atkin-Lehner involution; we have $j(W_N(\tau)) = j(N\tau)$ and the function f also satisfies:

- for canonical type: $f(W_N(\tau)) = N^s/f(\tau)$;
- for Atkin type: $f(W_N(\tau)) = f(\tau)$.

Furthermore, for an equation of canonical type, f is the standard η-quotient

$$f(\tau) = N^s(\eta(N\tau)/\eta(\tau))^{2s},$$

where η is Dedekind's eta function, which is invariant under $\Gamma_0(N)$.

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.

factorial (*x*, *precision=0*)
Factorial of x. The expression x! gives a result which is an integer, while $factorial(x)$ gives a real number.

fibonacci (*x*)
$x - th$ Fibonacci number.

galoisgetpol (*a*, *b=0*, *s=1*)
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 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.

getabstime ( )
Returns the CPU time (in milliseconds) elapsed since `gp` startup. This provides a reentrant version of `gettime` :

```
my (t = getabstime());
...
print("Time: ", getabstime() - t);
```

For a version giving wall-clock time, see `getwalltime` .

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.

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 the UNIX Epoch (1970-01-01 00:00:00 (UTC)).

```
my (t = getwalltime());
...
print("Time: ", getwalltime() - t);
```

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=None*, *lib=None*)

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 non-static 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`, `v` : return type int / long / 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`), `s` (`char *`), `n` (variable).
- `p` to supply `realprecision` (usually `long prec` in the argument list), `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 `lGD&`.

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.

intnumgaussinit (*n=0*, *precision=0*)

Initialize tables for n-point Gauss-Legendre integration of a smooth function f lon a compact interval $[a, b]$ at current `realprecision`. If n is omitted, make a default choice $n\ realprecision$, suitable for analytic functions on $[-1, 1]$. The error is bounded by

$$((b-a)^{2n+1}(n!)^4)/((2n+1)[(2n)!]^3)f^{(2n)}(\xi), a < \xi < b$$

so, if the interval length increases, n should be increased as well.

```
? 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
%3 = -9.490148553624725335 E-22

? 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`.

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);
```

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. 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.

mathilbert (*n*)
x being a `long` , creates the Hilbert matrixof order x, i.e. the matrix whose coefficient (i,:math:j) is $1/(i + j - 1)$.

matid (*n*)
Creates the nxn identity matrix.

matpascal (*n*, *q=None*)
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).

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 `permtonum` . The numbering used is the standard lexicographic ordering, starting at 0.

oo ( )
Returns an object meaning $+oo$, for use in functions such as `intnum` . 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.

partitions (*k*, *a=None*, *n=None*)
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 $amin <= 0$, in which case $nmin$ is ignored and X is of constant length $nmax$:

```
? 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])]
```

polchebyshev (*n*, *flag=1*, *a=None*)
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 $n >= 0$, 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, $U_{-1} = 0$.

polcyclo (*n*, *a=None*)

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 $\Phi_n(x) = \prod_{d\|n}(x^d - 1)^{\mu(n/d)}$. In the evaluated case, the algorithm assumes that $a^d - 1$ is either 0 or invertible, for all $d\|n$. If this is not the case (the base ring has zero divisors), use `subst(polcyclo(n),x,a)` .

polhermite (*n*, *a=None*)

$n - th$ Hermite polynomial H_n evaluated at a ('x by default), i.e.

$$H_n(x) = (-1)^n e^{x^2}(d^n)/(dx^n)e^{-x^2}.$$

pollegendre (*n*, *a=None*)

$n - th$ Legendre polynomial evaluated at a ('x by default).

polmodular (*L*, *inv=0*, *x=None*, *y=None*, *derivs=0*)

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 non-zero, 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 + ...
? 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*j + 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
```

polsubcyclo (*n*, *d*, *v=None*)

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.

`poltchebi` (*n*, *v=None*)
Deprecated alias for `polchebyshev`

`polylog` (*m*, *x*, *flag=0*, *precision=0*)
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 $\|x\| <= 1$ by

$$\Re_m(\sum_{k=0}^{m-1} ((-\log\|x\|)^k)/(k!)Li_{m-k}(x) + ((-\log\|x\|)^{m-1})/(m!)\log\|1-x\|).$$

If $flag = 2$: compute $D_m(x)$, defined for $\|x\| <= 1$ by

$$\Re_m(\sum_{k=0}^{m-1} ((-\log\|x\|)^k)/(k!)Li_{m-k}(x) - (1)/(2)((-\log\|x\|)^m)/(m!)).$$

If $flag = 3$: compute $P_m(x)$, defined for $\|x\| <= 1$ 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 $f_m(1/x) = (-1)^{m-1} f_m(x)$.

`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

$$2P_n^{(m)}(sin^2 t) = (d^{m+1})/(dt^{m+1})(\sin(2t)^m \sin(2(n-m)t)).$$

`prime` (*n*)
The $n - th$ prime number

```
? prime(10^9)
%1 = 22801763489
```

Uses checkpointing and a naive $O(n)$ algorithm.

`read` (*filename=None*)
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 `allocatemem` statement (to be generally avoided), you should leave `read` instructions by themselves, and not part of larger instruction sequences.

readstr (*filename=None*)
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=None*)
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.

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
```

stirling (*n, k, flag=1*)
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 non-negative 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 non-empty 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]));
}
```

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.

varhigher (*name*, *v=None*)

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`.

```
? Pol([x,x], t)
 *** at top-level: Pol([x,x],t)
 *** ^------------
 *** Pol: incorrect priority in gtopoly: variable x <= t
? t = varhigher("t", x);
? Pol([x,x], t)
%3 = x*t + x
```

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++)
```

varlower (*name*, *v=None*)

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.

```
\\ quickly use up all variables
? n = 0; while(1,varlower("x"); n++)
 *** at top-level: n=0;while(1,varlower("x");n++)
 *** ^-------------------
 *** varlower: 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,varlower("tmp", x); n++)
```

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`)!

`sage.libs.pari.pari_instance.`**default_bitprec** ( )

Return the default precision in bits.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import default_bitprec
sage: default_bitprec()
64
```

sage.libs.pari.pari_instance.**prec_bits_to_dec**(*prec_in_bits*)

Convert from precision expressed in bits to precision expressed in decimal.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_bits_to_dec
sage: prec_bits_to_dec(53)
15
sage: [(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)]
```

sage.libs.pari.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:

```
sage: from sage.libs.pari.pari_instance import prec_bits_to_words
sage: prec_bits_to_words(70)
5   # 32-bit
4   # 64-bit
```

```
sage: [(32*n, prec_bits_to_words(32*n)) for n in range(1, 9)]
[(32, 3), (64, 4), (96, 5), (128, 6), (160, 7), (192, 8), (224, 9), (256, 10)] #
↪32-bit
[(32, 3), (64, 3), (96, 4), (128, 4), (160, 5), (192, 5), (224, 6), (256, 6)] #
↪64-bit
```

sage.libs.pari.pari_instance.**prec_dec_to_bits**(*prec_in_dec*)

Convert from precision expressed in decimal to precision expressed in bits.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_dec_to_bits
sage: prec_dec_to_bits(15)
49
sage: [(n, prec_dec_to_bits(n)) for n in range(10, 100, 10)]
[(10, 33),
(20, 66),
(30, 99),
(40, 132),
(50, 166),
(60, 199),
(70, 232),
(80, 265),
(90, 298)]
```

`sage.libs.pari.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.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_dec_to_words
sage: prec_dec_to_words(38)
6   # 32-bit
4   # 64-bit
sage: [(n, prec_dec_to_words(n)) for n in range(10, 90, 10)]
[(10, 4), (20, 5), (30, 6), (40, 7), (50, 8), (60, 9), (70, 10), (80, 11)] # 32-
↪bit
[(10, 3), (20, 4), (30, 4), (40, 5), (50, 5), (60, 6), (70, 6), (80, 7)] # 64-bit
```

`sage.libs.pari.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:

```
sage: from sage.libs.pari.pari_instance import prec_words_to_bits
sage: prec_words_to_bits(10)
256   # 32-bit
512   # 64-bit
sage: [(n, prec_words_to_bits(n)) for n in range(3, 10)]
[(3, 32), (4, 64), (5, 96), (6, 128), (7, 160), (8, 192), (9, 224)]  # 32-bit
[(3, 64), (4, 128), (5, 192), (6, 256), (7, 320), (8, 384), (9, 448)] # 64-bit
```

`sage.libs.pari.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.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_words_to_dec
sage: prec_words_to_dec(5)
28   # 32-bit
57   # 64-bit
sage: [(n, prec_words_to_dec(n)) for n in range(3, 10)]
[(3, 9), (4, 19), (5, 28), (6, 38), (7, 48), (8, 57), (9, 67)] # 32-bit
[(3, 19), (4, 38), (5, 57), (6, 77), (7, 96), (8, 115), (9, 134)] # 64-bit
```

CHAPTER

TWENTYNINE

CONVERT PYTHON FUNCTIONS TO PARI CLOSURES

AUTHORS:

- Jeroen Demeyer (2015-04-10): initial version, trac ticket #18052.

EXAMPLES:

```
sage: def the_answer():
....:     return 42
sage: f = pari(the_answer)
sage: f()
42

sage: cube = pari(lambda i: i^3)
sage: cube.apply(range(10))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

`sage.libs.pari.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:

```
sage: from sage.libs.pari.closure import objtoclosure
sage: mul = objtoclosure(lambda i,j: i*j)
sage: mul
(v1,v2,v3,v4,v5)->call_python(v1,v2,v3,v4,v5,...)
sage: mul.type()
't_CLOSURE'
sage: mul(6,9)
54
sage: def printme(x):
....:     print(x)
sage: objtoclosure(printme)('matid(2)')
[1, 0; 0, 1]
```

Test various kinds of errors:

```
sage: mul(4)
Traceback (most recent call last):
...
TypeError: <lambda>() takes exactly 2 arguments (1 given)
sage: mul(None, None)
Traceback (most recent call last):
...
ValueError: Cannot convert None to pari
sage: mul(*range(100))
Traceback (most recent call last):
...
PariError: call_python: too many parameters in user-defined function call
sage: mul([1], [2])
Traceback (most recent call last):
...
PariError: call_python: forbidden multiplication t_VEC (1 elts) * t_VEC (1 elts)
```

CHAPTER

THIRTY

CONVERT PARI OBJECTS TO/FROM PYTHON INTEGERS

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.

`sage.libs.pari.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:

```
sage: from sage.libs.pari.convert import gen_to_integer
sage: a = gen_to_integer(pari("12345")); a; type(a)
12345
<type 'int'>
sage: a = gen_to_integer(pari("10^30")); a; type(a)
1000000000000000000000000000000L
<type 'long'>
sage: gen_to_integer(pari("19/5"))
3
sage: gen_to_integer(pari("1 + 0.0*I"))
1
sage: gen_to_integer(pari("3/2 + 0.0*I"))
1
sage: gen_to_integer(pari("Mod(-1, 11)"))
10
sage: gen_to_integer(pari("5 + O(5^10)"))
5
sage: gen_to_integer(pari("Pol(42)"))
42
sage: gen_to_integer(pari("x"))
Traceback (most recent call last):
...
TypeError: unable to convert PARI object x of type t_POL to an integer
sage: gen_to_integer(pari("x + O(x^2)"))
Traceback (most recent call last):
...
```

```
TypeError: unable to convert PARI object x + O(x^2) of type t_SER to an integer
sage: 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:

```
sage: for i in range(10000):
....:     x = 3**i
....:     if long(pari(x)) != long(x):
....:         print(x)
sage: gen_to_integer(pari("1.0 - 2^64"))
-18446744073709551615L
sage: gen_to_integer(pari("1 - 2^64"))
-18446744073709551615L
```

Check some corner cases:

```
sage: 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))
```

sage.libs.pari.convert.**integer_to_gen**(*x*)

Convert a Python `int` or `long` to a PARI `gen` of type `t_INT`.

EXAMPLES:

```
sage: from sage.libs.pari.convert import integer_to_gen
sage: a = integer_to_gen(int(12345)); a; type(a)
12345
<type 'sage.libs.pari.gen.gen'>
sage: a = integer_to_gen(long(12345)); a; type(a)
12345
<type 'sage.libs.pari.gen.gen'>
sage: integer_to_gen(float(12345))
Traceback (most recent call last):
...
TypeError: integer_to_gen() needs an int or long argument, not float
```

TESTS:

```
sage: for i in range(10000):
....:     x = 3**i
....:     if pari(long(x)) != pari(x):
....:         print(x)
```

CHAPTER

THIRTYONE

FUNCTIONS FOR HANDLING PARI ERRORS

AUTHORS:

- Peter Bruin (September 2013): initial version (trac ticket #9640)
- Jeroen Demeyer (January 2015): use `cb_pari_err_handle` (trac ticket #14894)

exception `sage.libs.pari.handle_error.` **`PariError`**

Bases: `exceptions.RuntimeError`

Error raised by PARI

`errdata` ( )

Return the error data (a `t_ERROR` gen) corresponding to this error.

EXAMPLES:

```
sage: try:
....:     pari(Mod(2,6))^-1
....: except PariError as e:
....:     E = e.errdata()
sage: E
error("impossible inverse in Fp_inv: Mod(2, 6).")
sage: E.component(2)
Mod(2, 6)
```

`errnum` ( )

Return the PARI error number corresponding to this exception.

EXAMPLES:

```
sage: try:
....:     pari('1/0')
....: except PariError as err:
....:     print(err.errnum())
31
```

`errtext` ( )

Return the message output by PARI when this error occurred.

EXAMPLE:

```
sage: try:
....:     pari('pi()')
....: except PariError as e:
....:     print(e.errtext())
not a function in function call
```

CHAPTER

THIRTYTWO

RING OF PARI OBJECTS

AUTHORS:

- William Stein (2004): Initial version.
- Simon King (2011-08-24): Use UniqueRepresentation, element_class and proper initialisation of elements.

class `sage.rings.pari_ring.`**`Pari`** (*x*, *parent=None*)

Bases: `sage.structure.element.RingElement`

Element of Pari pseudo-ring.

class `sage.rings.pari_ring.`**`PariRing`**

Bases: `sage.misc.fast_methods.Singleton`, `sage.rings.ring.Ring`

EXAMPLES: sage: R = PariRing(); R Pseudoring of all PARI objects. sage: loads(R.dumps()) is R True

`Element`

alias of *`Pari`*

`characteristic` ( )

`is_field` (*proof=True*)

`random_element` (*x=None*, *y=None*, *distribution=None*)

Return a random integer in Pari.

NOTE:

The given arguments are passed to `ZZ.random_element(...)`.

INPUT:

- x, y – optional integers, that are lower and upper bound for the result. If only x is provided, then the result is between 0 and $x - 1$, inclusive. If both are provided, then the result is between x and $y - 1$, inclusive.
- $distribution$ – optional string, so that `ZZ` can make sense of it as a probability distribution.

EXAMPLE:

```
sage: R = PariRing()
sage: R.random_element()
-8
sage: R.random_element(5,13)
12
sage: [R.random_element(distribution="1/n") for _ in range(10)]
[0, 1, -1, 2, 1, -95, -1, -2, -12, 0]
```

zeta ( )

Return -1.

EXAMPLE:

```
sage: R = PariRing()
sage: R.zeta()
-1
```

CHAPTER

THIRTYTHREE

READLINE

This is the library behind the command line input, it takes keypresses until you hit Enter and then returns it as a string to Python. We hook into it so we can make it redraw the input area.

EXAMPLES:

```
sage: from sage.libs.readline import *
sage: replace_line('foobar', 0)
sage: set_point(3)
sage: print('current line: ' + repr(copy_text(0, get_end())))
current line: 'foobar'
sage: print('cursor position: {}'.format(get_point()))
cursor position: 3
```

When printing with *interleaved_output* the prompt and current line is removed:

```
sage: with interleaved_output():
....:     print('output')
....:     print('current line: ' + repr(copy_text(0, get_end())))
....:     print('cursor position: {}'.format(get_point()))
output
current line: ''
cursor position: 0
```

After the interleaved output, the line and cursor is restored to the old value:

```
sage: print('current line: ' + repr(copy_text(0, get_end())))
current line: 'foobar'
sage: print('cursor position: {}'.format(get_point()))
cursor position: 3
```

Finally, clear the current line for the remaining doctests:

```
sage: replace_line('', 1)
```

sage.libs.readline.**clear_signals**()

Remove the readline signal handlers

Remove all of the Readline signal handlers installed by *set_signals()*

EXAMPLES:

```
sage: from sage.libs.readline import clear_signals
sage: clear_signals()
0
```

`sage.libs.readline.` **`copy_text`** (*pos_start*, *pos_end*)

Return a copy of the text between start and end in the current line.

INPUT:

- `pos_start` , `pos_end` – integer. Start and end position.

OUTPUT:

String.

EXAMPLES:

```
sage: from sage.libs.readline import copy_text, replace_line
sage: replace_line('foobar', 0)
sage: copy_text(1, 5)
'ooba'
```

`sage.libs.readline.` **`forced_update_display`** ( )

Force the line to be updated and redisplayed, whether or not Readline thinks the screen display is correct.

EXAMPLES:

```
sage: from sage.libs.readline import forced_update_display
sage: forced_update_display()
0
```

`sage.libs.readline.` **`get_end`** ( )

Get the end position of the current input

OUTPUT:

Integer

EXAMPLES:

```
sage: from sage.libs.readline import get_end
sage: get_end()
0
```

`sage.libs.readline.` **`get_point`** ( )

Get the cursor position

OUTPUT:

Integer

EXAMPLES:

```
sage: from sage.libs.readline import get_point, set_point
sage: get_point()
0
sage: set_point(5)
sage: get_point()
5
sage: set_point(0)
```

`sage.libs.readline.` **`initialize`** ( )

Initialize or re-initialize Readline's internal state. It's not strictly necessary to call this; readline() calls it before reading any input.

EXAMPLES:

```
sage: from sage.libs.readline import initialize
sage: initialize()
0
```

class sage.libs.readline.**interleaved_output**

Context manager for asynchronous output

This allows you to show output while at the readline prompt. When the block is left, the prompt is restored even if it was clobbered by the output.

EXAMPLES:

```
sage: from sage.libs.readline import interleaved_output
sage: with interleaved_output():
....:     print('output')
output
```

sage.libs.readline.**print_status**()

Print readline status for debug purposes

EXAMPLES:

```
sage: from sage.libs.readline import print_status
sage: print_status()
catch_signals: 1
catch_sigwinch: 1
```

sage.libs.readline.**replace_line**(*text*, *clear_undo*)

Replace the contents of rl_line_buffer with text.

The point and mark are preserved, if possible.

INPUT:

- `text` – the new content of the line.
- `clear_undo` – integer. If non-zero, the undo list associated with the current line is cleared.

EXAMPLES:

```
sage: from sage.libs.readline import copy_text, replace_line
sage: replace_line('foobar', 0)
sage: copy_text(1, 5)
'ooba'
```

sage.libs.readline.**set_point**(*point*)

Set the cursor position

INPUT:

- `point` – integer. The new cursor position.

EXAMPLES:

```
sage: from sage.libs.readline import get_point, set_point
sage: get_point()
0
sage: set_point(5)
sage: get_point()
5
sage: set_point(0)
```

sage.libs.readline.**set_signals**()

Install the readline signal handlers

Install Readline's signal handler for SIGINT, SIGQUIT, SIGTERM, SIGALRM, SIGTSTP, SIGTTIN, SIGTTOU, and SIGWINCH, depending on the values of rl_catch_signals and rl_catch_sigwinch.

EXAMPLES:

```
sage: from sage.libs.readline import set_signals
sage: set_signals()
0
```

CHAPTER

THIRTYFOUR

CONTEXT MANAGERS FOR LIBGAP

This module implements a context manager for global variables. This is useful since the behavior of GAP is sometimes controlled by global variables, which you might want to switch to a different value for a computation. Here is an example how you are suppose to use it from your code. First, let us set a dummy global variable for our example:

```
sage: libgap.set_global('FooBar', 123)
```

Then, if you want to switch the value momentarily you can write:

```
sage: with libgap.global_context('FooBar', 'test'):
....:     print(libgap.get_global('FooBar'))
test
```

Afterward, the global variable reverts to the previous value:

```
sage: print(libgap.get_global('FooBar'))
123
```

The value is reset even if exceptions occur:

```
sage: with libgap.global_context('FooBar', 'test'):
....:     print(libgap.get_global('FooBar'))
....:     raise ValueError(libgap.get_global('FooBar'))
Traceback (most recent call last):
...
ValueError: test
sage: print(libgap.get_global('FooBar'))
123
```

class `sage.libs.gap.context_managers.`**`GlobalVariableContext`**`(`*variable*, *value*`)`

Context manager for GAP global variables.

It is recommended that you use the *`sage.libs.gap.libgap.Gap.global_context()`* method and not construct objects of this class manually.

INPUT:

- `variable` – string. The variable name.
- `value` – anything that defines a GAP object.

EXAMPLES:

```
sage: libgap.set_global('FooBar', 1)
sage: with libgap.global_context('FooBar', 2):
....:     print(libgap.get_global('FooBar'))
2
```

```
sage: libgap.get_global('FooBar')
1
```

CHAPTER

THIRTYFIVE

GAP FUNCTIONS

CHAPTER

THIRTYSIX

LONG TESTS FOR LIBGAP

These stress test the garbage collection inside GAP

sage.libs.gap.test_long.**test_loop_1**()

EXAMPLES:

```
sage: from sage.libs.gap.test_long import test_loop_1
sage: test_loop_1()  # long time (up to 25s on sage.math, 2013)
```

sage.libs.gap.test_long.**test_loop_2**()

EXAMPLES:

```
sage: from sage.libs.gap.test_long import test_loop_2
sage: test_loop_2()  # long time (10s on sage.math, 2013)
```

sage.libs.gap.test_long.**test_loop_3**()

EXAMPLES:

```
sage: from sage.libs.gap.test_long import test_loop_3
sage: test_loop_3()  # long time (31s on sage.math, 2013)
```

CHAPTER

THIRTYSEVEN

UTILITY FUNCTIONS FOR LIBGAP

class sage.libs.gap.util.**ObjWrapper**

Bases: object

Wrapper for GAP master pointers

EXAMPLES:

```
sage: from sage.libs.gap.util import ObjWrapper
sage: x = ObjWrapper()
sage: y = ObjWrapper()
sage: x == y
True
```

sage.libs.gap.util.**command**(*command_string*)

Playground for accessing Gap via libGap.

You should not use this function in your own programs. This is just here for convenience if you want to play with the libgap libray code.

EXAMPLES:

```
sage: from sage.libs.gap.util import command
sage: command('1')
Output follows...
1

sage: command('1/0')
Traceback (most recent call last):
...
ValueError: libGAP: Error, Rational operations: <divisor> must not be zero

sage: command('NormalSubgroups')
Output follows...
<Attribute "NormalSubgroups">

sage: command('rec(a:=1, b:=2)')
Output follows...
rec( a := 1, b := 2 )
```

sage.libs.gap.util.**error_enter_libgap_block_twice**()

Demonstrate that we catch errors from entering a block twice.

EXAMPLES:

```
sage: from sage.libs.gap.util import error_enter_libgap_block_twice
sage: error_enter_libgap_block_twice()
```

```
Traceback (most recent call last):
...
RuntimeError: Entered a critical block twice
```

sage.libs.gap.util.**error_exit_libgap_block_without_enter**()

Demonstrate that we catch errors from omitting libgap_enter.

EXAMPLES:

```
sage: from sage.libs.gap.util import error_exit_libgap_block_without_enter
sage: error_exit_libgap_block_without_enter()
Traceback (most recent call last):
...
RuntimeError: Called libgap_exit without previous libgap_enter
```

sage.libs.gap.util.**gap_root**()

Find the location of the GAP root install which is stored in the gap startup script.

EXAMPLES:

```
sage: from sage.libs.gap.util import gap_root
sage: gap_root()    # random output
'/home/vbraun/opt/sage-5.3.rc0/local/gap/latest'
```

sage.libs.gap.util.**get_owned_objects**()

Helper to access the refcount dictionary from Python code

sage.libs.gap.util.**memory_usage**()

Return information about the memory useage.

See *mem()* for details.

CHAPTER

THIRTYEIGHT

LIBGAP SHARED LIBRARY INTERFACE TO GAP

This module implements a fast C library interface to GAP. To use libGAP you simply call `libgap` (the parent of all *GapElement* instances) and use it to convert Sage objects into GAP objects.

EXAMPLES:

```
sage: a = libgap(10)
sage: a
10
sage: type(a)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: a*a
100
sage: timeit('a*a')   # random output
625 loops, best of 3: 898 ns per loop
```

Compared to the expect interface this is >1000 times faster:

```
sage: b = gap('10')
sage: timeit('b*b')   # random output; long time
125 loops, best of 3: 2.05 ms per loop
```

If you want to evaluate GAP commands, use the *Gap.eval()* method:

```
sage: libgap.eval('List([1..10], i->i^2)')
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

not to be confused with the `libgap` call, which converts Sage objects to GAP objects, for example strings to strings:

```
sage: libgap('List([1..10], i->i^2)')
"List([1..10], i->i^2)"
sage: type(_)
<type 'sage.libs.gap.element.GapElement_String'>
```

You can usually use the *sage()* method to convert the resulting GAP element back to its Sage equivalent:

```
sage: a.sage()
10
sage: type(_)
<type 'sage.rings.integer.Integer'>

sage: libgap.eval('5/3 + 7*E(3)').sage()
7*zeta3 + 5/3

sage: generators = libgap.AlternatingGroup(4).GeneratorsOfGroup().sage()
sage: generators   # a Sage list of Sage permutations!
```

```
[(1,2,3), (2,3,4)]
sage: PermutationGroup(generators).cardinality()    # computed in Sage
12
sage: libgap.AlternatingGroup(4).Size()             # computed in GAP
12
```

So far, the following GAP data types can be directly converted to the corresponding Sage datatype:

1. GAP booleans `true` / `false` to Sage booleans `True` / `False`. The third GAP boolean value `fail` raises a `ValueError`.
2. GAP integers to Sage integers.
3. GAP rational numbers to Sage rational numbers.
4. GAP cyclotomic numbers to Sage cyclotomic numbers.
5. GAP permutations to Sage permutations.
6. The GAP containers `List` and `rec` are converted to Sage containers `list` and `dict`. Furthermore, the `sage()` method is applied recursively to the entries.

Special support is available for the GAP container classes. GAP lists can be used as follows:

```
sage: lst = libgap([1,5,7]);  lst
[ 1, 5, 7 ]
sage: type(lst)
<type 'sage.libs.gap.element.GapElement_List'>
sage: len(lst)
3
sage: lst[0]
1
sage: [ x^2 for x in lst ]
[1, 25, 49]
sage: type(_[0])
<type 'sage.libs.gap.element.GapElement_Integer'>
```

Note that you can access the elements of GAP `List` objects as you would expect from Python (with indexing starting at 0), but the elements are still of type `GapElement`. The other GAP container type are records, which are similar to Python dictionaries. You can construct them directly from Python dictionaries:

```
sage: libgap({'a':123, 'b':456})
rec( a := 123, b := 456 )
```

Or get them as results of computations:

```
sage: rec = libgap.eval('rec(a:=123, b:=456, Sym3:=SymmetricGroup(3))')
sage: rec['Sym3']
Sym( [ 1 .. 3 ] )
sage: dict(rec)
{'Sym3': Sym( [ 1 .. 3 ] ), 'a': 123, 'b': 456}
```

The output is a Sage dictionary whose keys are Sage strings and whose Values are instances of `GapElement()`. So, for example, `rec['a']` is not a Sage integer. To recursively convert the entries into Sage objects, you should use the `sage()` method:

```
sage: rec.sage()
{'Sym3': NotImplementedError('cannot construct equivalent Sage object',),
 'a': 123,
 'b': 456}
```

Now `rec['a']` is a Sage integer. We have not implemented the conversion of the GAP symmetric group to the Sage symmetric group yet, so you end up with a `NotImplementedError` exception object. The exception is returned and not raised so that you can work with the partial result.

While we don't directly support matrices yet, you can convert them to Gap List of Lists. These lists are then easily converted into Sage using the recursive expansion of the *sage()* method:

```
sage: M = libgap.eval('BlockMatrix([[1,1,[[1, 2],[ 3, 4]]], [1,2,[[9,10],[11,12]]],␣
↪[2,2,[[5, 6],[ 7, 8]]]],2,2)')
sage: M
<block matrix of dimensions (2*2)x(2*2)>
sage: M.List()   # returns a GAP List of Lists
[ [ 1, 2, 9, 10 ], [ 3, 4, 11, 12 ], [ 0, 0, 5, 6 ], [ 0, 0, 7, 8 ] ]
sage: M.List().sage()   # returns a Sage list of lists
[[1, 2, 9, 10], [3, 4, 11, 12], [0, 0, 5, 6], [0, 0, 7, 8]]
sage: matrix(ZZ, _)
[ 1  2  9 10]
[ 3  4 11 12]
[ 0  0  5  6]
[ 0  0  7  8]
```

38.1 Using the libGAP C library from Cython

The lower-case `libgap_foobar` functions are ones that we added to make the libGAP C shared library. The `libGAP_foobar` methods are the original GAP methods simply prefixed with the string `libGAP_`. The latter were originally not designed to be in a library, so some care needs to be taken to call them.

In particular, you must call `libgap_mark_stack_bottom()` in every function that calls into the libGAP C functions. The reason is that the GAP memory manager will automatically keep objects alive that are referenced in local (stack-allocated) variables. While convenient, this requires to look through the stack to find anything that looks like an address to a memory bag. But this requires vigilance against the following pattern:

```
cdef f()
  libgap_mark_stack_bottom()
  libGAP_function()

cdef g()
  libgap_mark_stack_bottom();
  f()                #  f() changed the stack bottom marker
  libGAP_function()  #  boom
```

The solution is to re-order `g()` to first call `f()`. In order to catch this error, it is recommended that you wrap calls into libGAP in `libgap_enter` / `libgap_exit` blocks and not call `libgap_mark_stack_bottom` manually. So instead, always write

cdef f() libgap_enter() libGAP_function() libgap_exit()

cdef g() f() libgap_enter() libGAP_function() libgap_exit()

If you accidentally call `libgap_enter()` twice then an error message is printed to help you debug this:

```
sage: from sage.libs.gap.util import error_enter_libgap_block_twice
sage: error_enter_libgap_block_twice()
Traceback (most recent call last):
...
RuntimeError: Entered a critical block twice
```

AUTHORS:

- William Stein, Robert Miller (2009-06-23): first version
- Volker Braun, Dmitrii Pasechnik, Ivan Andrus (2011-03-25, Sage Days 29): almost complete rewrite; first usable version.
- Volker Braun (2012-08-28, GAP/Singular workshop): update to gap-4.5.5, make it ready for public consumption.

class `sage.libs.gap.libgap.`**`Gap`**

Bases: `sage.structure.parent.Parent`

The libgap interpreter object.

Note: This object must be instantiated exactly once by the libgap. Always use the provided `libgap` instance, and never instantiate *`Gap`* manually.

EXAMPLES:

```
sage: libgap.eval('SymmetricGroup(4)')
Sym( [ 1 .. 4 ] )
```

TESTS:

```
sage: TestSuite(libgap).run(skip=['_test_category', '_test_elements', '_test_
↪pickling'])
```

`Element`

alias of `GapElement`

`collect` ( )

Manually run the garbage collector

EXAMPLES:

```
sage: a = libgap(123)
sage: del a
sage: libgap.collect()
```

`count_GAP_objects` ( )

Return the number of GAP objects that are being tracked by libGAP

OUTPUT:

An integer

EXAMPLES:

```
sage: libgap.count_GAP_objects()    # random output
5
```

`eval` (*gap_command*)

Evaluate a gap command and wrap the result.

INPUT:

- `gap_command` – a string containing a valid gap command without the trailing semicolon.

OUTPUT:

A `GapElement`.

EXAMPLES:

```
sage: libgap.eval('0')
0
sage: libgap.eval('"string"')
"string"
```

function_factory (*function_name*)
Return a GAP function wrapper

This is almost the same as calling `libgap.eval(function_name)`, but faster and makes it obvious in your code that you are wrapping a function.

INPUT:

- `function_name` – string. The name of a GAP function.

OUTPUT:

A function wrapper *GapElement_Function* for the GAP function. Calling it from Sage is equivalent to calling the wrapped function from GAP.

EXAMPLES:

```
sage: libgap.function_factory('Print')
<Gap function "Print">
```

get_global (*variable*)
Get a GAP global variable

INPUT:

- `variable` – string. The variable name.

OUTPUT:

A *GapElement* wrapping the GAP output. A `ValueError` is raised if there is no such variable in GAP.

EXAMPLES:

```
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
...
ValueError: libGAP: Error, VAL_GVAR: No value bound to FooBar
```

global_context (*variable*, *value*)
Temporarily change a global variable

INPUT:

- `variable` – string. The variable name.
- `value` – anything that defines a GAP object.

OUTPUT:

A context manager that sets/reverts the given global variable.

EXAMPLES:

```
sage: libgap.set_global('FooBar', 1)
sage: with libgap.global_context('FooBar', 2):
....:     print(libgap.get_global('FooBar'))
2
sage: libgap.get_global('FooBar')
1
```

mem ( )

Return information about libGAP memory usage

The GAP workspace is partitioned into 5 pieces (see gasman.c in the GAP sources for more details):

- The **masterpointer area** contains all the masterpointers of the bags.
- The **old bags area** contains the bodies of all the bags that survived at least one garbage collection. This area is only scanned for dead bags during a full garbage collection.
- The **young bags area** contains the bodies of all the bags that have been allocated since the last garbage collection. This area is scanned for dead bags during each garbage collection.
- The **allocation area** is the storage that is available for allocation of new bags. When a new bag is allocated the storage for the body is taken from the beginning of this area, and this area is correspondingly reduced. If the body does not fit in the allocation area a garbage collection is performed.
- The **unavailable area** is the free storage that is not available for allocation.

OUTPUT:

This function returns a tuple containing 5 integers. Each is the size (in bytes) of the five partitions of the workspace. This will potentially change after each GAP garbage collection.

EXAMPLES:

```
sage: libgap.collect()
sage: libgap.mem()    # random output
(1048576, 6706782, 0, 960930, 0)

sage: libgap.FreeGroup(3)
<free group on the generators [ f1, f2, f3 ]>
sage: libgap.mem()    # random output
(1048576, 6706782, 47571, 913359, 0)

sage: libgap.collect()
sage: libgap.mem()    # random output
(1048576, 6734785, 0, 998463, 0)
```

one ( )

Return (integer) one in GAP.

EXAMPLES:

```
sage: libgap.one()
1
sage: parent(_)
C library interface to GAP
```

set_global (*variable*, *value*)

Set a GAP global variable

INPUT:

- `variable` – string. The variable name.
- `value` – anything that defines a GAP object.

EXAMPLES:

```
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
...
ValueError: libGAP: Error, VAL_GVAR: No value bound to FooBar
```

show ( )

Print statistics about the GAP owned object list

Slight complication is that we want to do it without accessing libgap objects, so we don't create new GapElements as a side effect.

EXAMPLES:

```
sage: a = libgap(123)
sage: b = libgap(456)
sage: c = libgap(789)
sage: del b
sage: libgap.show() # random output
11 LibGAP elements currently alive
rec( full := rec( cumulative := 122, deadbags := 9,
deadkb := 0, freekb := 7785, livebags := 304915,
livekb := 47367, time := 33, totalkb := 68608 ),
nfull := 3, npartial := 14 )
```

unset_global (*variable*)

Remove a GAP global variable

INPUT:

- `variable` – string. The variable name.

EXAMPLES:

```
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
...
ValueError: libGAP: Error, VAL_GVAR: No value bound to FooBar
```

zero ( )

Return (integer) zero in GAP.

OUTPUT:

A `GapElement`.

EXAMPLES:

```
sage: libgap.zero()
0
```

TESTS:

```
sage: libgap.zero_element()
doctest:...: DeprecationWarning: zero_element is deprecated. Please use zero␣
↪instead.
See http://trac.sagemath.org/17694 for details.
0
```

zero_element (**args*, ***kwds*)
Deprecated: Use `zero()` instead. See trac ticket #17694 for details.

CHAPTER

THIRTYNINE

SHORT TESTS FOR LIBGAP

sage.libs.gap.test.**test_write_to_file**()

Test that libgap can write to files

See trac ticket #16502, trac ticket #15833.

EXAMPLES:

```
sage: from sage.libs.gap.test import test_write_to_file
sage: test_write_to_file()
```

CHAPTER FORTY

LIBGAP ELEMENT WRAPPER

This document describes the individual wrappers for various GAP elements. For general information about libGAP, you should read the *libgap* module documentation.

class sage.libs.gap.element.**GapElement**

Bases: sage.structure.element.RingElement

Wrapper for all Gap objects.

Note: In order to create GapElements you should use the libgap instance (the parent of all Gap elements) to convert things into GapElement. You must not create GapElement instances manually.

EXAMPLES:

```
sage: libgap(0)
0
```

If Gap finds an error while evaluating, a corresponding assertion is raised:

```
sage: libgap.eval('1/0')
Traceback (most recent call last):
...
ValueError: libGAP: Error, Rational operations: <divisor> must not be zero
```

Also, a ValueError is raised if the input is not a simple expression:

```
sage: libgap.eval('1; 2; 3')
Traceback (most recent call last):
...
ValueError: can only evaluate a single statement
```

is_bool()

Return whether the wrapped GAP object is a GAP boolean.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: libgap(True).is_bool()
True
```

is_function()

Return whether the wrapped GAP object is a function.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: a = libgap.eval("NormalSubgroups")
sage: a.is_function()
True
sage: a = libgap(2/3)
sage: a.is_function()
False
```

is_list ( )

Return whether the wrapped GAP object is a GAP List.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: libgap.eval('[1, 2,,,, 5]').is_list()
True
sage: libgap.eval('3/2').is_list()
False
```

is_permutation ( )

Return whether the wrapped GAP object is a GAP permutation.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: perm = libgap.PermList( libgap([1,5,2,3,4]) );  perm
(2,5,4,3)
sage: perm.is_permutation()
True
sage: libgap('this is a string').is_permutation()
False
```

is_record ( )

Return whether the wrapped GAP object is a GAP record.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: libgap.eval('[1, 2,,,, 5]').is_record()
False
sage: libgap.eval('rec(a:=1, b:=3)').is_record()
True
```

is_string ( )

Return whether the wrapped GAP object is a GAP string.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: libgap('this is a string').is_string()
True
```

sage ( )

Return the Sage equivalent of the *GapElement*

EXAMPLES:

```
sage: libgap(1).sage()
1
sage: type(_)
<type 'sage.rings.integer.Integer'>

sage: libgap(3/7).sage()
3/7
sage: type(_)
<type 'sage.rings.rational.Rational'>

sage: libgap.eval('5 + 7*E(3)').sage()
7*zeta3 + 5

sage: libgap(Infinity).sage()
+Infinity
sage: libgap(-Infinity).sage()
-Infinity

sage: libgap(True).sage()
True
sage: libgap(False).sage()
False
sage: type(_)
<type 'bool'>

sage: libgap('this is a string').sage()
'this is a string'
sage: type(_)
<type 'str'>
```

class sage.libs.gap.element.**GapElement_Boolean**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP boolean values.

EXAMPLES:

```
sage: b = libgap(True)
sage: type(b)
<type 'sage.libs.gap.element.GapElement_Boolean'>
```

sage ( )

Return the Sage equivalent of the *GapElement*

OUTPUT:

A Python boolean if the values is either true or false. GAP booleans can have the third value `Fail`, in which case a `ValueError` is raised.

EXAMPLES:

```
sage: b = libgap.eval('true');  b
true
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Boolean'>
sage: b.sage()
True
sage: type(_)
<type 'bool'>

sage: libgap.eval('fail')
fail
sage: _.sage()
Traceback (most recent call last):
...
ValueError: the GAP boolean value "fail" cannot be represented in Sage
```

class sage.libs.gap.element.**GapElement_Cyclotomic**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP universal cyclotomics.

EXAMPLES:

```
sage: libgap.eval('E(3)')
E(3)
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Cyclotomic'>
```

sage (*ring=None*)

Return the Sage equivalent of the *GapElement_Cyclotomic*.

INPUT:

- `ring` – a Sage cyclotomic field or `None` (default). If not specified, a suitable minimal cyclotomic field will be constructed.

OUTPUT:

A Sage cyclotomic field element.

EXAMPLES:

```
sage: n = libgap.eval('E(3)')
sage: n.sage()
zeta3
sage: parent(_)
Cyclotomic Field of order 3 and degree 2

sage: n.sage(ring=CyclotomicField(6))
zeta6 - 1

sage: libgap.E(3).sage(ring=CyclotomicField(3))
zeta3
sage: libgap.E(3).sage(ring=CyclotomicField(6))
zeta6 - 1
```

TESTS:

Check that trac ticket #15204 is fixed:

```
sage: libgap.E(3).sage(ring=UniversalCyclotomicField())
E(3)
sage: libgap.E(3).sage(ring=CC)
-0.500000000000000 + 0.866025403784439*I
```

class sage.libs.gap.element.**GapElement_FiniteField**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP finite field elements.

EXAMPLES:

```
sage: libgap.eval('Z(5)^2')
Z(5)^2
sage: type(_)
<type 'sage.libs.gap.element.GapElement_FiniteField'>
```

lift ()

Return an integer lift.

OUTPUT:

The smallest positive *GapElement_Integer* that equals `self` in the prime finite field.

EXAMPLES:

```
sage: n = libgap.eval('Z(5)^2')
sage: n.lift()
4
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Integer'>

sage: n = libgap.eval('Z(25)')
sage: n.lift()
Traceback (most recent call last):
TypeError: not in prime subfield
```

sage (*ring=None*, *var='a'*)

Return the Sage equivalent of the *GapElement_FiniteField*.

INPUT:

- `ring` – a Sage finite field or `None` (default). The field to return `self` in. If not specified, a suitable finite field will be constructed.

OUTPUT:

An Sage finite field element. The isomorphism is chosen such that the Gap `PrimitiveRoot()` maps to the Sage `multiplicative_generator()`.

EXAMPLES:

```
sage: n = libgap.eval('Z(25)^2')
sage: n.sage()
a + 3
sage: parent(_)
Finite Field in a of size 5^2

sage: n.sage(ring=GF(5))
Traceback (most recent call last):
```

```
...
ValueError: the given finite field has incompatible size
```

class sage.libs.gap.element.**GapElement_Function**
Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP functions.

EXAMPLES:

```
sage: f = libgap.Cycles
sage: type(f)
<type 'sage.libs.gap.element.GapElement_Function'>
```

class sage.libs.gap.element.**GapElement_Integer**
Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP integers.

EXAMPLES:

```
sage: i = libgap(123)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: ZZ(i)
123
```

is_C_int()
Return whether the wrapped GAP object is a immediate GAP integer.

An immediate integer is one that is stored as a C integer, and is subject to the usual size limits. Larger integers are stored in GAP as GMP integers.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: n = libgap(1)
sage: type(n)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: n.is_C_int()
True
sage: n.IsInt()
true

sage: N = libgap(2^130)
sage: type(N)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: N.is_C_int()
False
sage: N.IsInt()
true
```

sage(*ring=None*)
Return the Sage equivalent of the *GapElement_Integer*

- `ring` – Integer ring or `None` (default). If not specified, a the default Sage integer ring is used.

OUTPUT:

A Sage integer

EXAMPLES:

```
sage: libgap([ 1, 3, 4 ]).sage()
[1, 3, 4]
sage: all( x in ZZ for x in _ )
True

sage: libgap(132).sage(ring=IntegerModRing(13))
2
sage: parent(_)
Ring of integers modulo 13
```

TESTS:

```
sage: large = libgap.eval('2^130');  large
1361129467683753853853498429727072845824
sage: large.sage()
1361129467683753853853498429727072845824

sage: huge = libgap.eval('10^9999');  huge     # gap abbreviates very long␣
↪ints
<integer 100...000 (10000 digits)>
sage: huge.sage().ndigits()
10000
```

class sage.libs.gap.element.**GapElement_IntegerMod**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP integers modulo an integer.

EXAMPLES:

```
sage: n = IntegerModRing(123)(13)
sage: i = libgap(n)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_IntegerMod'>
```

lift()

Return an integer lift.

OUTPUT:

A *GapElement_Integer* that equals `self` in the integer mod ring.

EXAMPLES:

```
sage: n = libgap.eval('One(ZmodnZ(123)) * 13')
sage: n.lift()
13
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Integer'>
```

sage(*ring=None*)

Return the Sage equivalent of the *GapElement_IntegerMod*

INPUT:

- `ring` – Sage integer mod ring or `None` (default). If not specified, a suitable integer mod ringa is used automatically.

OUTPUT:

A Sage integer modulo another integer.

EXAMPLES:

```
sage: n = libgap.eval('One(ZmodnZ(123)) * 13')
sage: n.sage()
13
sage: parent(_)
Ring of integers modulo 123
```

class sage.libs.gap.element.**GapElement_List**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP Lists.

Note: Lists are indexed by $0..len(l) - 1$, as expected from Python. This differs from the GAP convention where lists start at 1.

EXAMPLES:

```
sage: lst = libgap.SymmetricGroup(3).List(); lst
[ (), (1,3), (1,2,3), (2,3), (1,3,2), (1,2) ]
sage: type(lst)
<type 'sage.libs.gap.element.GapElement_List'>
sage: len(lst)
6
sage: lst[3]
(2,3)
```

We can easily convert a Gap `List` object into a Python `list`:

```
sage: list(lst)
[(), (1,3), (1,2,3), (2,3), (1,3,2), (1,2)]
sage: type(_)
<type 'list'>
```

Range checking is performed:

```
sage: lst[10]
Traceback (most recent call last):
...
IndexError: index out of range.
```

matrix (*ring=None*)

Return the list as a matrix.

GAP does not have a special matrix data type, they are just lists of lists. This function converts a GAP list of lists to a Sage matrix.

OUTPUT:

A Sage matrix.

EXAMPLES:

```
sage: F = libgap.GF(4)
sage: a = F.PrimitiveElement()
```

```
sage: m = libgap([[a,a^0],[0*a,a^2]]); m
[ [ Z(2^2), Z(2)^0 ],
  [ 0*Z(2), Z(2^2)^2 ] ]
sage: m.IsMatrix()
true
sage: matrix(m)
[    a     1]
[    0 a + 1]
sage: matrix(GF(4,'B'), m)
[    B     1]
[    0 B + 1]

sage: M = libgap.eval('SL(2,GF(5))').GeneratorsOfGroup()[1]
sage: type(M)
<type 'sage.libs.gap.element.GapElement_List'>
sage: M[0][0]
Z(5)^2
sage: M.IsMatrix()
true
sage: M.matrix()
[4 1]
[4 0]
```

sage (***kwds*)

Return the Sage equivalent of the *GapElement*

OUTPUT:

A Python list.

EXAMPLES:

```
sage: libgap([ 1, 3, 4 ]).sage()
[1, 3, 4]
sage: all( x in ZZ for x in _ )
True
```

vector (*ring=None*)

Return the list as a vector.

GAP does not have a special vetor data type, they are just lists. This function converts a GAP list to a Sage vector.

OUTPUT:

A Sage vector.

EXAMPLES:

```
sage: F = libgap.GF(4)
sage: a = F.PrimitiveElement()
sage: m = libgap([0*a, a, a^3, a^2]); m
[ 0*Z(2), Z(2^2), Z(2)^0, Z(2^2)^2 ]
sage: type(m)
<type 'sage.libs.gap.element.GapElement_List'>
sage: m[3]
Z(2^2)^2
sage: vector(m)
(0, a, 1, a + 1)
```

```
sage: vector(GF(4,'B'), m)
(0, B, 1, B + 1)
```

class sage.libs.gap.element.**GapElement_MethodProxy**

Bases: *sage.libs.gap.element.GapElement_Function*

Helper class returned by `GapElement.__getattr__`.

Derived class of GapElement for GAP functions. Like its parent, you can call instances to implement function call syntax. The only difference is that a fixed first argument is prepended to the argument list.

EXAMPLES:

```
sage: lst = libgap([])
sage: lst.Add
<Gap function "Add">
sage: type(_)
<type 'sage.libs.gap.element.GapElement_MethodProxy'>
sage: lst.Add(1)
sage: lst
[ 1 ]
```

class sage.libs.gap.element.**GapElement_Permutation**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP permutations.

Note: Permutations in GAP act on the numbers starting with 1.

EXAMPLES:

```
sage: perm = libgap.eval('(1,5,2)(4,3,8)')
sage: type(perm)
<type 'sage.libs.gap.element.GapElement_Permutation'>
```

sage ()

Return the Sage equivalent of the *GapElement*

EXAMPLES:

```
sage: perm_gap = libgap.eval('(1,5,2)(4,3,8)');  perm_gap
(1,5,2)(3,8,4)
sage: perm_gap.sage()
(1,5,2)(3,8,4)
sage: type(_)
<type 'sage.groups.perm_gps.permgroup_element.PermutationGroupElement'>
```

class sage.libs.gap.element.**GapElement_Rational**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP rational numbers.

EXAMPLES:

```
sage: r = libgap(123/456)
sage: type(r)
<type 'sage.libs.gap.element.GapElement_Rational'>
```

sage (*ring=None*)

Return the Sage equivalent of the *GapElement* .

INPUT:

- ring – the Sage rational ring or None (default). If not specified, the rational ring is used automatically.

OUTPUT:

A Sage rational number.

EXAMPLES:

```
sage: r = libgap(123/456);  r
41/152
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Rational'>
sage: r.sage()
41/152
sage: type(_)
<type 'sage.rings.rational.Rational'>
```

class sage.libs.gap.element. **GapElement_Record**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP records.

EXAMPLES:

```
sage: rec = libgap.eval('rec(a:=123, b:=456)')
sage: type(rec)
<type 'sage.libs.gap.element.GapElement_Record'>
sage: len(rec)
2
sage: rec['a']
123
```

We can easily convert a Gap rec object into a Python dict :

```
sage: dict(rec)
{'a': 123, 'b': 456}
sage: type(_)
<type 'dict'>
```

Range checking is performed:

```
sage: rec['no_such_element']
Traceback (most recent call last):
...
IndexError: libGAP: Error, Record: '<rec>.no_such_element' must have an assigned␣
↪value
```

record_name_to_index (*py_name*)

Convert string to GAP record index.

INPUT:

- py_name – a python string.

OUTPUT:

A `UInt`, which is a GAP hash of the string. If this is the first time the string is encountered, a new integer is returned(!)

EXAMPLE:

```
sage: rec = libgap.eval('rec(first:=123, second:=456)')
sage: rec.record_name_to_index('first')   # random output
1812L
sage: rec.record_name_to_index('no_such_name') # random output
3776L
```

sage ( )

Return the Sage equivalent of the *GapElement*

EXAMPLES:

```
sage: libgap.eval('rec(a:=1, b:=2)').sage()
{'a': 1, 'b': 2}
sage: all( isinstance(key,str) and val in ZZ for key,val in _.items() )
True

sage: rec = libgap.eval('rec(a:=123, b:=456, Sym3:=SymmetricGroup(3))')
sage: rec.sage()
{'Sym3': NotImplementedError('cannot construct equivalent Sage object',),
 'a': 123,
 'b': 456}
```

class sage.libs.gap.element.**GapElement_RecordIterator**

Bases: object

Iterator for *GapElement_Record*

Since Cython does not support generators yet, we implement the older iterator specification with this auxiliary class.

INPUT:

- rec – the *GapElement_Record* to iterate over.

EXAMPLES:

```
sage: rec = libgap.eval('rec(a:=123, b:=456)')
sage: list(rec)
[('a', 123), ('b', 456)]
sage: dict(rec)
{'a': 123, 'b': 456}
```

next ( )

x.next() -> the next value, or raise StopIteration

class sage.libs.gap.element.**GapElement_Ring**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP rings (parents of ring elements).

EXAMPLES:

```
sage: i = libgap(ZZ)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_Ring'>
```

ring_cyclotomic ( )

Construct an integer ring.

EXAMPLES:

```
sage: libgap.CyclotomicField(6).ring_cyclotomic()
Cyclotomic Field of order 3 and degree 2
```

ring_finite_field (*var='a'*)

Construct an integer ring.

EXAMPLES:

```
sage: libgap.GF(3,2).ring_finite_field(var='A')
Finite Field in A of size 3^2
```

ring_integer ( )

Construct the Sage integers.

EXAMPLES:

```
sage: libgap.eval('Integers').ring_integer()
Integer Ring
```

ring_integer_mod ( )

Construct a Sage integer mod ring.

EXAMPLES:

```
sage: libgap.eval('ZmodnZ(15)').ring_integer_mod()
Ring of integers modulo 15
```

ring_rational ( )

Construct the Sage rationals.

EXAMPLES:

```
sage: libgap.eval('Rationals').ring_rational()
Rational Field
```

sage (***kwds*)

Return the Sage equivalent of the `GapElement_Ring`.

INPUT:

- `**kwds` – keywords that are passed on to the `ring_` method.

OUTPUT:

A Sage ring.

EXAMPLES:

```
sage: libgap.eval('Integers').sage()
Integer Ring

sage: libgap.eval('Rationals').sage()
Rational Field

sage: libgap.eval('ZmodnZ(15)').sage()
Ring of integers modulo 15
```

```
sage: libgap.GF(3,2).sage(var='A')
Finite Field in A of size 3^2

sage: libgap.CyclotomicField(6).sage()
Cyclotomic Field of order 3 and degree 2
```

class sage.libs.gap.element.**GapElement_String**

Bases: *sage.libs.gap.element.GapElement*

Derived class of GapElement for GAP strings.

EXAMPLES:

```
sage: s = libgap('string')
sage: type(s)
<type 'sage.libs.gap.element.GapElement_String'>
sage: s
"string"
sage: print(s)
string
```

sage ( )

Convert this *GapElement_String* to a Python string.

OUTPUT:

A Python string.

EXAMPLES:

```
sage: s = libgap.eval(' "string" '); s
"string"
sage: type(_)
<type 'sage.libs.gap.element.GapElement_String'>
sage: str(s)
'string'
sage: s.sage()
'string'
sage: type(_)
<type 'str'>
```

CHAPTER

FORTYONE

LIBGAP WORKSPACE SUPPORT

The single purpose of this module is to provide the location of the libgap saved workspace and a time stamp to invalidate saved workspaces.

`sage.libs.gap.saved_workspace.`**`timestamp`**`()`

Return a time stamp for libgap

OUTPUT:

Float. Unix timestamp of the most recently changed LibGAP file.

EXAMPLES:

```
sage: from sage.libs.gap.saved_workspace import timestamp
sage: timestamp()    # random output
1406642467.25684
sage: type(timestamp())
<type 'float'>
```

`sage.libs.gap.saved_workspace.`**`workspace`**`(`*name='workspace'*`)`

Return the filename of the gap workspace and whether it is up to date.

INPUT:

- `name` – string. A name that will become part of the workspace filename.

OUTPUT:

Pair consisting of a string and a boolean. The string is the filename of the saved libgap workspace (or that it should have if it doesn't exist). The boolean is whether the workspace is up-to-date. You may use the workspace file only if the boolean is `True`.

EXAMPLES:

```
sage: from sage.libs.gap.saved_workspace import workspace
sage: ws, up_to_date = workspace()
sage: ws
'/.../gap/libgap-workspace-...'
sage: isinstance(up_to_date, bool)
True
```

CHAPTER

FORTYTWO

LIBRARY INTERFACE TO EMBEDDABLE COMMON LISP (ECL)

class sage.libs.ecl.**EclListIterator**

Bases: object

Iterator object for an ECL list

This class is used to implement the iterator protocol for EclObject. Do not instantiate this class directly but use the iterator method on an EclObject instead. It is an error if the EclObject is not a list.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: I=EclListIterator(EclObject("(1 2 3)"))
sage: type(I)
<type 'sage.libs.ecl.EclListIterator'>
sage: [i for i in I]
[<ECL: 1>, <ECL: 2>, <ECL: 3>]
sage: [i for i in EclObject("(1 2 3)")]
[<ECL: 1>, <ECL: 2>, <ECL: 3>]
sage: EclListIterator(EclObject("1"))
Traceback (most recent call last):
...
TypeError: ECL object is not iterable
```

next ( )

x.next() -> the next value, or raise StopIteration

class sage.libs.ecl.**EclObject**

Bases: object

Python wrapper of ECL objects

The EclObject forms a wrapper around ECL objects. The wrapper ensures that the data structure pointed to is protected from garbage collection in ECL by installing a pointer to it from a global data structure within the scope of the ECL garbage collector. This pointer is destroyed upon destruction of the EclObject.

EclObject() takes a Python object and tries to find a representation of it in Lisp.

EXAMPLES:

Python lists get mapped to LISP lists. None and Boolean values to appropriate values in LISP:

```
sage: from sage.libs.ecl import *
sage: EclObject([None,true,false])
<ECL: (NIL T NIL)>
```

Numerical values are translated to the appropriate type in LISP:

```
sage: EclObject(1)
<ECL: 1>
sage: EclObject(10**40)
<ECL: 10000000000000000000000000000000000000000>
```

Floats in Python are IEEE double, which LISP has as well. However, the printing of floating point types in LISP depends on settings:

```
sage: a = EclObject(float(10^40))
sage: ecl_eval("(setf *read-default-float-format* 'single-float)")
<ECL: SINGLE-FLOAT>
sage: a
<ECL: 1.d40>
sage: ecl_eval("(setf *read-default-float-format* 'double-float)")
<ECL: DOUBLE-FLOAT>
sage: a
<ECL: 1.e40>
```

Tuples are translated to dotted lists:

```
sage: EclObject( (false, true))
<ECL: (NIL . T)>
```

Strings are fed to the reader, so a string normally results in a symbol:

```
sage: EclObject("Symbol")
<ECL: SYMBOL>
```

But with proper quotation one can construct a lisp string object too:

```
sage: EclObject('"Symbol"')
<ECL: "Symbol">
```

EclObjects translate to themselves, so one can mix:

```
sage: EclObject([1,2,EclObject([3])])
<ECL: (1 2 (3))>
```

Calling an EclObject translates into the appropriate LISP `apply` , where the argument is transformed into an EclObject itself, so one can flexibly apply LISP functions:

```
sage: car=EclObject("car")
sage: cdr=EclObject("cdr")
sage: car(cdr([1,2,3]))
<ECL: 2>
```

and even construct and evaluate arbitrary S-expressions:

```
sage: eval=EclObject("eval")
sage: quote=EclObject("quote")
sage: eval([car, [cdr, [quote,[1,2,3]]]])
<ECL: 2>
```

TESTS:

We check that multiprecision integers are converted correctly:

```
sage: i = 10 ^ (10 ^ 5)
sage: EclObject(i) == EclObject(str(i))
True
sage: EclObject(-i) == EclObject(str(-i))
True
sage: EclObject(i).python() == i
True
sage: EclObject(-i).python() == -i
True
```

atomp ( )

Return True if self is atomic, False otherwise.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).atomp()
True
sage: EclObject([[]]).atomp()
False
```

caar ( )

Return the caar of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cadr ( )

Return the cadr of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
```

```
sage: L.cddr()
<ECL: NIL>
```

car ( )

Return the car of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cdar ( )

Return the cdar of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cddr ( )

Return the cddr of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
```

```
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cdr ( )

Return the cdr of self

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

characterp ( )

Return True if self is a character, False otherwise

Strings are not characters

EXAMPLES:

sage: from sage.libs.ecl import * sage: EclObject("'a"').characterp() False

cons (*d*)

apply cons to self and argument and return the result.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: a=EclObject(1)
sage: b=EclObject(2)
sage: a.cons(b)
<ECL: (1 . 2)>
```

consp ( )

Return True if self is a cons, False otherwise. NIL is not a cons.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).consp()
False
sage: EclObject([[]]).consp()
True
```

eval ( )

Evaluate object as an S-Expression

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: S=EclObject("(+ 1 2)")
sage: S
<ECL: (+ 1 2)>
sage: S.eval()
<ECL: 3>
```

fixnump ( )

Return True if self is a fixnum, False otherwise

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject(2**3).fixnump()
True
sage: EclObject(2**200).fixnump()
False
```

listp ( )

Return True if self is a list, False otherwise. NIL is a list.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).listp()
True
sage: EclObject([[]]).listp()
True
```

nullp ( )

Return True if self is NIL, False otherwise

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).nullp()
True
sage: EclObject([[]]).nullp()
False
```

python ( )

Convert an EclObject to a python object.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([1,2,("three",'"four"')])
sage: L.python()
[1, 2, ('THREE', '"four"')]
```

rplaca (*d*)

Destructively replace car(self) with d.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject((1,2))
sage: L
<ECL: (1 . 2)>
```

```
sage: a=EclObject(3)
sage: L.rplaca(a)
sage: L
<ECL: (3 . 2)>
```

rplacd (*d*)

Destructively replace cdr(self) with d.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject((1,2))
sage: L
<ECL: (1 . 2)>
sage: a=EclObject(3)
sage: L.rplacd(a)
sage: L
<ECL: (1 . 3)>
```

symbolp ( )

Return True if self is a symbol, False otherwise.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).symbolp()
True
sage: EclObject([[]]).symbolp()
False
```

sage.libs.ecl.**ecl_eval** (*s*)

Read and evaluate string in Lisp and return the result

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: ecl_eval("(defun fibo (n)(cond((= n 0) 0)((= n 1) 1)(T (+ (fibo (- n 1))␣
↪(fibo (- n 2))))))")
<ECL: FIBO>
sage: ecl_eval("(mapcar 'fibo '(1 2 3 4 5 6 7))")
<ECL: (1 1 2 3 5 8 13)>
```

sage.libs.ecl.**init_ecl** ( )

Internal function to initialize ecl. Do not call.

This function initializes the ECL library for use within Python. This routine should only be called once and importing the ecl library interface already does that, so do not call this yourself.

EXAMPLES:

```
sage: from sage.libs.ecl import *
```

At this point, init_ecl() has run. Explicitly executing it gives an error:

```
sage: init_ecl()
Traceback (most recent call last):
...
RuntimeError: ECL is already initialized
```

sage.libs.ecl.**print_objects**()

Print GC-protection list

Diagnostic function. ECL objects that are bound to Python objects need to be protected from being garbage collected. We do this by including them in a doubly linked list bound to the global ECL symbol *SAGE-LIST-OF-OBJECTS*. Only non-immediate values get included, so small integers do not get linked in. This routine prints the values currently stored.

EXAMPLE:

```
sage: from sage.libs.ecl import *
sage: a=EclObject("hello")
sage: b=EclObject(10)
sage: c=EclObject("world")
sage: print_objects() #random because previous test runs can have left objects
NIL
WORLD
HELLO
```

sage.libs.ecl.**shutdown_ecl**()

Shut down ecl. Do not call.

Given the way that ECL is used from python, it is very difficult to ensure that no ECL objects exist at a particular time. Hence, destroying ECL is a risky proposition.

EXAMPLE:

```
sage: from sage.libs.ecl import *
sage: shutdown_ecl()
```

sage.libs.ecl.**test_ecl_options**()

Print an overview of the ECL options

TESTS:

```
sage: from sage.libs.ecl import test_ecl_options
sage: test_ecl_options()
ECL_OPT_INCREMENTAL_GC = 0
ECL_OPT_TRAP_SIGSEGV = 1
ECL_OPT_TRAP_SIGFPE = 1
ECL_OPT_TRAP_SIGINT = 1
ECL_OPT_TRAP_SIGILL = 1
ECL_OPT_TRAP_SIGBUS = 1
ECL_OPT_TRAP_SIGCHLD = 0
ECL_OPT_TRAP_SIGPIPE = 1
ECL_OPT_TRAP_INTERRUPT_SIGNAL = 1
ECL_OPT_SIGNAL_HANDLING_THREAD = 0
ECL_OPT_SIGNAL_QUEUE_SIZE = 16
ECL_OPT_BOOTED = 1
ECL_OPT_BIND_STACK_SIZE = ...
ECL_OPT_BIND_STACK_SAFETY_AREA = ...
ECL_OPT_FRAME_STACK_SIZE = ...
ECL_OPT_FRAME_STACK_SAFETY_AREA = ...
ECL_OPT_LISP_STACK_SIZE = ...
ECL_OPT_LISP_STACK_SAFETY_AREA = ...
ECL_OPT_C_STACK_SIZE = ...
ECL_OPT_C_STACK_SAFETY_AREA = ...
ECL_OPT_SIGALTSTACK_SIZE = 1
ECL_OPT_HEAP_SIZE = ...
ECL_OPT_HEAP_SAFETY_AREA = ...
```

```
ECL_OPT_THREAD_INTERRUPT_SIGNAL = 0
ECL_OPT_SET_GMP_MEMORY_FUNCTIONS = 0
```

sage.libs.ecl.**test_sigint_before_ecl_sig_on**()

TESTS:

If an interrupt arrives *before* ecl_sig_on(), we should get an ordinary KeyboardInterrupt:

```
sage: from sage.libs.ecl import test_sigint_before_ecl_sig_on
sage: test_sigint_before_ecl_sig_on()
KeyboardInterrupt
Traceback (most recent call last):
...
```

CHAPTER

FORTYTHREE

GSL ARRAYS

class `sage.gsl.gsl_array.` **`GSLDoubleArray`**

Bases: `object`

EXAMPLES:

```
sage: a = WaveletTransform(128,'daubechies',4)
sage: for i in range(1, 11):
....:     a[i] = 1
sage: a[:6:2]
[0.0, 1.0, 1.0]
```

CHAPTER

FORTYFOUR

INTERFACE TO THE PSELECT() SYSTEM CALL

This module defines a class `PSelecter` which can be used to call the system call `pselect()` and which can also be used in a `with` statement to block given signals until `PSelecter.pselect()` is called.

AUTHORS:

- Jeroen Demeyer (2013-02-07): initial version (trac ticket #14079)

44.1 Waiting for subprocesses

One possible use is to wait with a **timeout** until **any child process** exits, as opposed to `os.wait()` which doesn't have a timeout or `multiprocessing.Process.join()` which waits for one specific process.

Since `SIGCHLD` is ignored by default, we first need to install a signal handler for `SIGCHLD`. It doesn't matter what it does, as long as the signal isn't ignored:

```
sage: import signal
sage: def dummy_handler(sig, frame):
....:     pass
....:
sage: _ = signal.signal(signal.SIGCHLD, dummy_handler)
```

We wait for a child created using the `subprocess` module:

```
sage: from sage.ext.pselect import PSelecter
sage: from subprocess import *
sage: with PSelecter([signal.SIGCHLD]) as sel:
....:     p = Popen(["sleep", "1"])
....:     _ = sel.sleep()
....:
sage: p.poll()  # p should be finished
0
```

Now using the `multiprocessing` module:

```
sage: from sage.ext.pselect import PSelecter
sage: from multiprocessing import *
sage: import time
sage: with PSelecter([signal.SIGCHLD]) as sel:
....:     p = Process(target=time.sleep, args=(1,))
....:     p.start()
....:     _ = sel.sleep()
....:     p.is_alive()  # p should be finished
```

```
....:
False
```

class sage.ext.pselect.**PSelecter**

Bases: object

This class gives an interface to the pselect system call.

It can be used in a with statement to block given signals such that they can only occur during the *pselect()* or *sleep()* calls.

As an example, we block the SIGHUP and SIGALRM signals and then raise a SIGALRM signal. The interrupt will only be seen during the *sleep()* call:

```
sage: from sage.ext.pselect import PSelecter
sage: import signal, time
sage: with PSelecter([signal.SIGHUP, signal.SIGALRM]) as sel:
....:     os.kill(os.getpid(), signal.SIGALRM)
....:     time.sleep(0.5)  # Simply sleep, no interrupt detected
....:     try:
....:         _ = sel.sleep(1)  # Interrupt seen here
....:     except AlarmInterrupt:
....:         print("Interrupt OK")
....:
Interrupt OK
```

Warning: If SIGCHLD is blocked inside the with block, then you should not use Popen().wait() or Process().join() because those might block, even if the process has actually exited. Use non-blocking alternatives such as Popen.poll() or multiprocessing.active_children() instead.

pselect(*rlist=[], wlist=[], xlist=[], timeout=None*)

Wait until one of the given files is ready, or a signal has been received, or until timeout seconds have past.

INPUT:

- rlist – (default: []) a list of files to wait for reading.
- wlist – (default: []) a list of files to wait for writing.
- xlist – (default: []) a list of files to wait for exceptions.
- timeout – (default: None) a timeout in seconds, where None stands for no timeout.

OUTPUT: A 4-tuple (rready,wready,xready,tmout) where the first three are lists of file descriptors which are ready, that is a subset of (rlist,wlist,xlist) . The fourth is a boolean which is True if and only if the command timed out. If pselect was interrupted by a signal, the output is ([],[],[],False) .

See also:

Use the *sleep()* method instead if you don't care about file descriptors.

EXAMPLES:

The file /dev/null should always be available for reading and writing:

```
sage: from sage.ext.pselect import PSelecter
sage: f = open(os.devnull, "r+")
sage: sel = PSelecter()
sage: sel.pselect(rlist=[f])
([<open file '/dev/null', mode 'r+' at ...>], [], [], False)
sage: sel.pselect(wlist=[f])
([], [<open file '/dev/null', mode 'r+' at ...>], [], False)
```

A list of various files, all of them should be ready for reading. Also create a pipe, which should be ready for writing, but not reading (since nothing has been written):

```
sage: f = open(os.devnull, "r")
sage: g = open(os.path.join(SAGE_LOCAL, 'bin', 'python'), "r")
sage: (pr, pw) = os.pipe()
sage: r, w, x, t = PSelecter().pselect([f,g,pr,pw], [pw], [pr,pw])
sage: len(r), len(w), len(x), t
(2, 1, 0, False)
```

Checking for exceptions on the pipe should simply time out:

```
sage: sel.pselect(xlist=[pr,pw], timeout=0.2)
([], [], [], True)
```

TESTS:

It is legal (but silly) to list the same file multiple times:

```
sage: r, w, x, t = PSelecter().pselect([f,g,f,f,g])
sage: len(r)
5
```

Invalid input:

```
sage: PSelecter().pselect([None])
Traceback (most recent call last):
...
TypeError: an integer is required
```

Open a file and close it, but save the (invalid) file descriptor:

```
sage: f = open(os.devnull, "r")
sage: n = f.fileno()
sage: f.close()
sage: PSelecter().pselect([n])
Traceback (most recent call last):
...
IOError: ...
```

sleep (*timeout=None*)

Wait until a signal has been received, or until `timeout` seconds have past.

This is implemented as a special case of *pselect()* with empty lists of file descriptors.

INPUT:

- `timeout` – (default: `None`) a timeout in seconds, where `None` stands for no timeout.

OUTPUT: A boolean which is `True` if the call timed out, False if it was interrupted.

EXAMPLES:

A simple wait with timeout:

```
sage: from sage.ext.pselect import PSelecter
sage: sel = PSelecter()
sage: sel.sleep(timeout=0.1)
True
```

0 or negative time-outs are allowed, `sleep` should then return immediately:

```
sage: sel.sleep(timeout=0)
True
sage: sel.sleep(timeout=-123.45)
True
```

`sage.ext.pselect.`**`get_fileno`**`(`*f*`)`

Return the file descriptor of `f`.

INPUT:

- `f` – an object with a `.fileno` method or an integer, which is a file descriptor.

OUTPUT: A C `long` representing the file descriptor.

EXAMPLES:

```
sage: from sage.ext.pselect import get_fileno
sage: get_fileno(open(os.devnull))  # random
5
sage: get_fileno(42)
42
sage: get_fileno(None)
Traceback (most recent call last):
...
TypeError: an integer is required
sage: get_fileno(-1)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
sage: get_fileno(2^30)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
```

CHAPTER

FORTYFIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

e

g

l

r

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z