C/C++ Library Interfaces

Release 10.4

The Sage Development Team

Jul 20, 2024

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. Sage provides access to many libraries 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., `pexpect`), since everything is linked together and run as a single process. This is much more robust and efficient than using `pexpect`.

Each of these interfaces is used by other parts of Sage. For example, eclib 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

ECL

1.1 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)
<class '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
```

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(1.234e40))
sage: ecl_eval("(setf *read-default-float-format* 'single-float)")
<ECL: SINGLE-FLOAT>
sage: a
<ECL: 1.234d40>
sage: ecl_eval("(setf *read-default-float-format* 'double-float)")
<ECL: DOUBLE-FLOAT>
sage: a
<ECL: 1.234e40>
```

Tuples are translated to dotted lists:

```
sage: EclObject( (false, true))
<ECL: (NIL . T)>
sage: EclObject( (1, 2, 3) )
<ECL: (1 2 . 3)>
```

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">
```

Or any other object that the Lisp reader can construct:

```
sage: EclObject('#("I" am "just" a "simple" vector)')
<ECL: #("I" AM "just" A "simple" VECTOR)>
```

By means of Lisp reader macros, you can include arbitrary objects:

```
sage: EclObject([ 1, 2, '''#.(make-hash-table :test #'equal)''', 4])
<ECL: (1 2 #<hash-table ...> 4)>
```

Using an optional argument, you can control how strings are handled:

```
sage: EclObject("String", False)
<ECL: "String">
sage: EclObject('#(I may look like a vector but I am a string)', False)
<ECL: "#(I may look like a vector but I am a string)">
```

This also affects strings within nested lists and tuples

```
sage: EclObject([1, 2, "String", 4], False)
<ECL: (1 2 "String" 4)>
```

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>
```

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()
```

(continues on next page)

(continued from previous page)

```
<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.

EXAMPLES:

```
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.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: shutdown_ecl()
```

`sage.libs.ecl.`**`test_ecl_options`**`()`

Print an overview of the ECL options

`sage.libs.ecl.`**`test_sigint_before_ecl_sig_on`**`()`

CHAPTER

TWO

ECLIB

2.1 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`.

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

Bases: `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()
0.051111408239968840
```

saturate(*bound=-1*, *lower=2*)

Compute the saturation of the Mordell-Weil group.

INPUT:

- `bound` (int, default -1) – If -1, saturate at *all* primes by computing a bound on the saturation index, otherwise saturate at all primes up to the minimum of `bound` and the saturation index bound.
- `lower` (int, default 2) – Only saturate at primes not less than this.

EXAMPLES:

Since the 2-descent automatically saturates at primes up to 20, further saturation often has no 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 Issue #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-10
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.

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

Bases: `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 **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).

EXAMPLES:

```
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
```

(continues on next page)

(continued from previous page)

```
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...Saturation index bound (for points of good reduction)  = 3
Reducing saturation bound from given value 20 to computed index bound 3
Tamagawa index primes are [ 2 ]...
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 7)
Checking 3-saturation
Points were proved 3-saturated (max q used = 7)
done
P2 = [-2:3:1]     is generator number 2
saturating up to 20...Saturation index bound (for points of good reduction)  = 4
Reducing saturation bound from given value 20 to computed index bound 4
Tamagawa index primes are [ 2 ]...
Checking saturation at [ 2 3 ]
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]
Reducing index bound from 4 to 2
Points have successfully been 2-saturated (max q used = 7)
Index gain = 2^1
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...Saturation index bound (for points of good reduction)  = 3
Reducing saturation bound from given value 20 to computed index bound 3
Tamagawa index primes are [ 2 ]...
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)
```

(continues on next page)

(continued from previous page)

```
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*, *saturation_bound=0*)

Process points in the list v.

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

INPUT:

- v (list of 3-tuples or lists of ints or Integers) – a list of triples of integers, which define points on the curve.
- saturation_bound (int, default 0) – saturate at primes up to saturation_bound, or at *all* primes if saturation_bound is -1; when saturation_bound is 0 (the default), do no saturation..

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 `saturation_bound`:

```
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]], saturation_
↪bound=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]], saturation_
↪bound=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()
375.42920288254555
sage: EQ.saturate(2)  # points were not 2-saturated
saturating basis...Saturation index bound (for points of good reduction)  = 93
Only p-saturating for p up to given value 2.
The resulting points may not be p-saturated for p between this and the␣
↪computed index bound 93
Tamagawa index primes are [ 2 ]...
Checking saturation at [ 2 ]
Checking 2-saturation
possible kernel vector = [1,0,0]
This point may be in 2E(Q): [1547:-2967:343]
```

(continues on next page)

(continued from previous page)

```
...and it is!
Replacing old generator #1 with new generator [-2:3:1]
Reducing index bound from 93 to 46
Points have successfully been 2-saturated (max q used = 11)
Index gain = 2^1
done
Gained index 2
New regulator =  93.85730072
(True, 2, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [2707496766203306, 864581029138191, 2969715140223272], [-
↪13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
93.85730072063639
sage: EQ.saturate(3)  # points were not 3-saturated
saturating basis...Saturation index bound (for points of good reduction)  = 46
Only p-saturating for p up to given value 3.
The resulting points may not be p-saturated for p between this and the␣
↪computed index bound 46
Tamagawa index primes are [ 2 ]...
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
possible kernel vector = [0,1,0]
This point may be in 3E(Q):␣
↪[2707496766203306:864581029138191:2969715140223272]
...and it is!
Replacing old generator #2 with new generator [-14:25:8]
Reducing index bound from 46 to 15
Points have successfully been 3-saturated (max q used = 13)
Index gain = 3^1
done
Gained index 3
New regulator =  10.42858897
(True, 3, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
10.4285889689596
sage: EQ.saturate(5)  # points were not 5-saturated
saturating basis...Saturation index bound (for points of good reduction)  = 15
Only p-saturating for p up to given value 5.
The resulting points may not be p-saturated for p between this and the␣
↪computed index bound 15
Tamagawa index primes are [ 2 ]...
Checking saturation at [ 2 3 5 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
Checking 5-saturation
possible kernel vector = [0,0,1]
This point may be in 5E(Q): [-13422227300:-49322830557:12167000000]
...and it is!
Replacing old generator #3 with new generator [1:-1:1]
Reducing index bound from 15 to 3
```

(continues on next page)

(continued from previous page)

```
Points have successfully been 5-saturated (max q used = 71)
Index gain = 5^1
done
Gained index 5
New regulator =  0.4171435588
(True, 5, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [1, -1, 1]]
sage: EQ.regulator()
0.417143558758384
sage: EQ.saturate()   # points are now saturated
saturating basis...Saturation index bound (for points of good reduction)  = 3
Tamagawa index primes are [ 2 ]...
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)
```

(continues on next page)

(continued from previous page)

```
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
sage: EQ.rank()
3
sage: EQ.regulator()
0.417143558758384
```

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*, *min_prime=2*)

Saturate this subgroup of the Mordell-Weil group.

INPUT:

- `max_prime` (int, default -1) – If -1 (the default), an upper bound is computed for the primes at which the subgroup may not be saturated, and saturation is performed for all primes up to this bound. Otherwise, the bound used is the minimum of `max_prime` and the computed bound.
- `min_prime` (int, default 2) – only do saturation at primes no less than this. (For example, if the points have been found via `two_descent()` they should already be 2-saturated so a value of 3 is appropriate.)

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`, 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.

Note: In versions up to v20190909, `eclib` used floating point methods based on elliptic logarithms to divide points, and did not compute the precision necessary, which could cause failures. Since v20210310, `eclib` uses exact method based on division polynomials, which should mean that such failures does not happen.

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 computes an upper bound for the index of saturation, and does no work for primes greater than this even if `max_prime` is larger.

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]], saturation_
↪bound=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()
375.42920288254555
```

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

```
sage: EQ.saturate(2)  # points were not 2-saturated
saturating basis...Saturation index bound (for points of good reduction) = 93
Only p-saturating for p up to given value 2.
...
Gained index 2
New regulator =  93.857...
(True, 2, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1],␣
↪[2707496766203306:864581029138191:2969715140223272], [-13422227300:-
↪49322830557:12167000000]]
sage: EQ.regulator()
93.85730072063639
```

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

```
sage: EQ.saturate(3)  # points were not 3-saturated
saturating basis...Saturation index bound (for points of good reduction) = 46
Only p-saturating for p up to given value 3.
...
Gained index 3
New regulator =  10.428...
(True, 3, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [-13422227300:-
↪49322830557:12167000000]]
```

(continues on next page)

(continued from previous page)

```
sage: EQ.regulator()
10.4285889689596
```

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

```
sage: EQ.saturate(5)  # points were not 5-saturated
saturating basis...Saturation index bound (for points of good reduction) = 15
Only p-saturating for p up to given value 5.
...
Gained index 5
New regulator =  0.417...
(True, 5, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.417143558758384
```

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 (for points of good reduction) = 3
Tamagawa index primes are [ 2 ]...
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]], saturation_
↪bound=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()
0.417143558758384
```

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 (for points of good reduction) = 3
Tamagawa index primes are [ 2 ]...
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 ($31\log(2)$) else $\exp(h_{\text{lim}}) > 2^{31}$ and overflows. On 64-bit machines, it must be *at most* 43.668 ($63\log(2)$) . 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]]
```

2.2 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`**`()`

Return the working floating point bit precision of mwrank, which is equal to the global NTL real number precision.

OUTPUT:

(int) The current precision in bits.

See also *`set_precision()`*.

EXAMPLES:

```
sage: mwrank_get_precision()
150
```

`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: import tempfile
sage: with tempfile.NamedTemporaryFile(mode='w+t') as f:
....:     data = ' '.join(str(p) for p in prime_range(10^7, 10^7 + 20))
....:     _ = f.write(data)
....:     f.flush()
....:     mwrank_initprimes(f.name, 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(f.name, True)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
OSError: No such file or directory: ...
```

sage.libs.eclib.mwrank.**parse_point_list**(*s*)

Parse a string representing a list of points.

INPUT:

- s (string) – string representation of a list of points, for example '[]', '[[1:2:3]]', or '[[1:2:3],[4:5:6]]'.

OUTPUT:

(list) a list of triples of integers, for example [], [[1,2,3]], [[1,2,3],[4,5,6]].

EXAMPLES:

```
sage: from sage.libs.eclib.mwrank import parse_point_list
sage: parse_point_list('[]')
[]
sage: parse_point_list('[[1:2:3]]')
[[1, 2, 3]]
sage: parse_point_list('[[1:2:3],[4:5:6]]')
[[1, 2, 3], [4, 5, 6]]
```

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

Sets the working floating point bit precision of mwrank, which is equal to the global NTL real number precision.

NTL real number bit precision. This has a massive effect on the speed of mwrank calculations. The default (used if this function is not called) is n=150, but it might have to be increased if a computation fails.

INPUT:

- n – a positive integer: the number of bits of precision.

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

Note: The minimal value to which the precision may be set is 53. Lower values will be increased to 53.

See also *get_precision()*.

EXAMPLES:

```
sage: from sage.libs.eclib.mwrank import set_precision, get_precision
sage: old_prec = get_precision(); old_prec
150
sage: set_precision(50)
sage: get_precision()
53
sage: set_precision(old_prec)
sage: get_precision()
150
```

2.3 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)
<class 'sage.libs.eclib.mat.Matrix'>
sage: t
61 x 61 Cremona matrix over Rational Field
```

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)
<class '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)
<class '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`

2.4 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
```

By default, symbols are based at the cusp ∞, i.e. we evaluate $\{\infty, r\}$:

```
sage: [M(1/i) for i in range(1,11)]
[2/5, -8/5, -3/5, 7/5, 12/5, 12/5, 7/5, -3/5, -8/5, 2/5]
```

We can also switch the base point to the cusp 0:

```
sage: [M(1/i, base_at_infinity=False) for i in range(1,11)]
[0, -2, -1, 1, 2, 2, 1, -1, -2, 0]
```

For the minus symbols this makes no difference since $\{0, \infty\}$ is in the plus space. Note that to evaluate minus symbols the space must be defined with sign 0, which makes both signs available:

```
sage: M = ECModularSymbol(E,0); M
Modular symbol with sign 0 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, -1) for i in range(1,11)]
[0, 0, 1, 1, 0, 0, -1, -1, 0, 0]
sage: [M(1/i, -1, base_at_infinity=False) for i in range(1,11)]
[0, 0, 1, 1, 0, 0, -1, -1, 0, 0]
```

If the ECModularSymbol is created with sign 0 then as well as asking for both + and - symbols, we can also obtain both (as a tuple). However it is more work to create the full modular symbol space:

```
sage: E = EllipticCurve('11a1')
sage: M = ECModularSymbol(E,0); M
Modular symbol with sign 0 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(2,11)]
[[-8/5, 0],
 [-3/5, 1],
 [7/5, 1],
 [12/5, 0],
 [12/5, 0],
 [7/5, -1],
 [-3/5, -1],
 [-8/5, 0],
 [2/5, 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
```

Non-optimal curves are handled correctly in eclib, by comparing the ratios of real and/or imaginary periods:

```
sage: from sage.libs.eclib.newforms import ECModularSymbol
sage: E1 = EllipticCurve('11a1') # optimal
sage: E1.period_lattice().basis()
```

(continues on next page)

(continued from previous page)

```
(1.26920930427955, 0.634604652139777 + 1.45881661693850*I)
sage: M1 = ECModularSymbol(E1,0)
sage: M1(0)
[2/5, 0]
sage: M1(1/3)
[-3/5, 1]
```

One non-optimal curve has real period 1/5 that of the optimal one, so plus symbols scale up by a factor of 5 while minus symbols are unchanged:

```
sage: E2 = EllipticCurve('11a2') # not optimal
sage: E2.period_lattice().basis()
(0.253841860855911, 0.126920930427955 + 1.45881661693850*I)
sage: M2 = ECModularSymbol(E2,0)
sage: M2(0)
[2, 0]
sage: M2(1/3)
[-3, 1]
sage: all((M2(r,1)==5*M1(r,1)) for r in QQ.range_by_height(10))
True
sage: all((M2(r,-1)==M1(r,-1)) for r in QQ.range_by_height(10))
True
```

The other non-optimal curve has real period 5 times that of the optimal one, so plus symbols scale down by a factor of 5; again, minus symbols are unchanged:

```
sage: E3 = EllipticCurve('11a3') # not optimal
sage: E3.period_lattice().basis()
(6.34604652139777, 3.17302326069888 + 1.45881661693850*I)
sage: M3 = ECModularSymbol(E3,0)
sage: M3(0)
[2/25, 0]
sage: M3(1/3)
[-3/25, 1]
sage: all((5*M3(r,1)==M1(r,1)) for r in QQ.range_by_height(10))
True
sage: all((M3(r,-1)==M1(r,-1)) for r in QQ.range_by_height(10))
True
```

2.5 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)
<class '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
```

sparse_hecke_matrix(*p*, *dual=False*, *verbose=False*, *base_ring='ZZ'*)

Return the matrix of the `p`-th Hecke operator acting on this space of modular symbols as a sparse Sage matrix over `base_ring`. This is more memory-efficient than creating a Cremona matrix and then applying sage_matrix_over_ZZ with sparse=True.

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.sparse_hecke_matrix(2); type(t)
<class 'sage.matrix.matrix_integer_sparse.Matrix_integer_sparse'>
sage: print(t)
[ 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: M = CremonaModularSymbols(5001)
sage: T = M.sparse_hecke_matrix(2)
sage: U = M.hecke_matrix(2).sage_matrix_over_ZZ(sparse=True)
sage: print(T == U)
True
sage: T = M.sparse_hecke_matrix(2, dual=True)
sage: print(T == U.transpose())
True
sage: T = M.sparse_hecke_matrix(2, base_ring=GF(7))
sage: print(T == U.change_ring(GF(7)))
True
```

This concerns an issue reported on Issue #21303:

```
sage: C = CremonaModularSymbols(45, cuspidal=True,sign=-1)
sage: T2a = C.hecke_matrix(2).sage_matrix_over_ZZ()
sage: T2b = C.sparse_hecke_matrix(2)
sage: print(T2a == T2b)
True
```

2.6 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␣
```

(continues on next page)

(continued from previous page)

```
↪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 Issue #9476):

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

CHAPTER THREE

FLINT

3.1 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_sage.**Fmpz_poly**

Bases: 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_sage 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_sage 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_sage 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_sage 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_sage 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_sage 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_sage 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_sage 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_sage 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
```

3.2 File: sage/libs/flint/fmpq_poly_sage.pyx (starting at line 1)

3.3 FLINT Arithmetic Functions

sage.libs.flint.arith_sage.**bell_number**(*n*)

Return the n-th Bell number.

See Wikipedia article Bell_number.

ALGORITHM:

Uses `arith_bell_number()`.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage 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_sage.**bernoulli_number**(*n*)

Return the n-th Bernoulli number.

See Wikipedia article Bernoulli_number.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage 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_sage.**dedekind_sum**(p, q)

Return the Dedekind sum $s(p, q)$ where p and q are arbitrary integers.

See Wikipedia article Dedekind_sum.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage import dedekind_sum
sage: dedekind_sum(4, 5)
-1/5
```

sage.libs.flint.arith_sage.**euler_number**(n)

Return the Euler number of index n.

See Wikipedia article Euler_number.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage import euler_number
sage: [euler_number(i) for i in range(8)]
[1, 0, -1, 0, 5, 0, -61, 0]
```

sage.libs.flint.arith_sage.**harmonic_number**(n)

Return the harmonic number H_n.

See Wikipedia article Harmonic_number.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage 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_sage.**number_of_partitions**(n)

Return the number of partitions of the integer n.

See Wikipedia article Partition_(number_theory).

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage import number_of_partitions
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
```

(continues on next page)

(continued from previous page)

```
37338
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
274935105697756965126775163209863526881734293159800547582031259843021473281149641730550507416607
```

sage.libs.flint.arith_sage.**stirling_number_1**(*n*, *k*)

Return the unsigned Stirling number of the first kind.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage import stirling_number_1
sage: [stirling_number_1(8,i) for i in range(9)]
[0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1]
```

sage.libs.flint.arith_sage.**stirling_number_2**(*n*, *k*)

Return the Stirling number of the second kind.

EXAMPLES:

```
sage: from sage.libs.flint.arith_sage import stirling_number_2
sage: [stirling_number_2(8,i) for i in range(9)]
[0, 1, 127, 966, 1701, 1050, 266, 28, 1]
```

3.4 Interface to FLINT's `qsieve_factor()`. This used to interact

with an external "QuadraticSieve" program, but its functionality has been absorbed into flint.

sage.libs.flint.qsieve_sage.**qsieve**(*n*)

Factor `n` using the quadratic sieve.

INPUT:

- `n` – an integer; neither prime nor a perfect power.

OUTPUT:

A list of the factors of `n`. There is no guarantee that the factors found will be prime, or distinct.

EXAMPLES:

```
sage: k = 19; n = next_prime(10^k)*next_prime(10^(k+1))
sage: factor(n)  # (currently) uses PARI
10000000000000000051 * 100000000000000000039
sage: qsieve(n)
[(10000000000000000051, 1), (100000000000000000039, 1)]
```

3.5 File: sage/libs/flint/ulong_extras_sage.pyx (starting at line 1)

sage.libs.flint.ulong_extras_sage.**n_factor_to_list**(*n*, *proved*)

A wrapper around `n_factor`.

EXAMPLES:

```
sage: from sage.libs.flint.ulong_extras_sage import n_factor_to_list
sage: n_factor_to_list(60, 20)
[(2, 2), (3, 1), (5, 1)]
sage: n_factor_to_list((10**6).next_prime() + 1, 0)
[(2, 2), (53, 2), (89, 1)]
```

CHAPTER FOUR

GIAC

4.1 Wrappers for Giac functions

We provide a python function to compute and convert to sage a Groebner basis using the `giacpy_sage` module.

AUTHORS:

- Martin Albrecht (2015-07-01): initial version
- Han Frederic (2015-07-01): initial version

EXAMPLES:

```
sage: from sage.libs.giac import groebner_basis as gb_giac # random
sage: P = PolynomialRing(QQ, 6, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: B = gb_giac(I.gens()) # random
sage: B
Polynomial Sequence with 45 Polynomials in 6 Variables
```

class `sage.libs.giac.`**`GiacSettingsDefaultContext`**

Bases: `object`

Context preserve libgiac settings.

`sage.libs.giac.`**`groebner_basis`**(*gens*, *proba_epsilon=None*, *threads=None*, *prot=False*, *elim_variables=None*, **args*, ***kwds*)

Compute a Groebner Basis of an ideal using `giacpy_sage`. The result is automatically converted to sage.

Supported term orders of the underlying polynomial ring are `lex`, `deglex`, `degrevlex` and block orders with 2 `degrevlex` blocks.

INPUT:

- `gens` – an ideal (or a list) of polynomials over a prime field of characteristic 0 or p<2^31
- **`proba_epsilon`** **– (default: None) majoration of the probability**
  of a wrong answer when probabilistic algorithms are allowed.
  - if `proba_epsilon` is None, the value of `sage.structure.proof.all.polynomial()` is taken. If it is false then the global `giacpy_sage.giacsettings.proba_epsilon` is used.
  - if `proba_epsilon` is 0, probabilistic algorithms are disabled.
- `threads` – (default: None) Maximal number of threads allowed for giac. If None, the global `giacpy_sage.giacsettings.threads` is considered.

- `prot` – (default: `False`) if True print detailled informations
- `elim_variables` – (default: None) a list of variables to eliminate from the ideal.
  - if `elim_variables` is None, a Groebner basis with respect to the term ordering of the parent polynomial ring of the polynomials `gens` is computed.
  - if `elim_variables` is a list of variables, a Groebner basis of the elimination ideal with respect to a `degrevlex` term order is computed, regardless of the term order of the polynomial ring.

OUTPUT:

Polynomial sequence of the reduced Groebner basis.

EXAMPLES:

```
sage: from sage.libs.giac import groebner_basis as gb_giac
sage: P = PolynomialRing(GF(previous_prime(2**31)), 6, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: B = gb_giac(I.gens())
...
sage: B
Polynomial Sequence with 45 Polynomials in 6 Variables
sage: B.is_groebner()
True
```

Elimination ideals can be computed by passing `elim_variables`:

```
sage: P = PolynomialRing(GF(previous_prime(2**31)), 5, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: B = gb_giac(I.gens(), elim_variables=[P.gen(0), P.gen(2)])
...
sage: B.is_groebner()
True
sage: B.ideal() == I.elimination_ideal([P.gen(0), P.gen(2)])
True
```

Computations over QQ can benefit from

- a probabilistic lifting:

```
sage: P = PolynomialRing(QQ,5, 'x')
sage: I = ideal([P.random_element(3,7) for j in range(5)])
sage: B1 = gb_giac(I.gens(),1e-16) # long time (1s)
...
sage: sage.structure.proof.all.polynomial(True)
sage: B2 = gb_giac(I.gens()) # long time (4s)
...
sage: B1 == B2 # long time
True
sage: B1.is_groebner() # not tested, too long time (50s)
True
```

- multi threaded operations:

```
sage: P = PolynomialRing(QQ, 8, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: time B = gb_giac(I.gens(),1e-6,threads=2) # doctest: +SKIP
...
Time: CPU 168.98 s, Wall: 94.13 s
```

You can get detailled information by setting `prot=True`

```
sage: I = sage.rings.ideal.Katsura(P)
sage: gb_giac(I,prot=True)  # random, long time (3s)
9381383 begin computing basis modulo 535718473
9381501 begin new iteration zmod, number of pairs: 8, base size: 8
...end, basis size 74 prime number 1
G=Vector [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...
...creating reconstruction #0
...
++++++++basis size 74
checking pairs for i=0, j=
checking pairs for i=1, j=2,6,12,17,19,24,29,34,39,42,43,48,56,61,64,69,
...
checking pairs for i=72, j=73,
checking pairs for i=73, j=
Number of critical pairs to check 373
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++...
Successful... check of 373 critical pairs
12380865 end final check
Polynomial Sequence with 74 Polynomials in 8 Variables
```

`sage.libs.giac.`**`local_giacsettings`**`(`*func*`)`

Decorator to preserve Giac's proba_epsilon and threads settings.

EXAMPLES:

```
sage: def testf(a,b):
....:     giacsettings.proba_epsilon = a/100
....:     giacsettings.threads = b+2
....:     return (giacsettings.proba_epsilon, giacsettings.threads)

sage: from sage.libs.giac.giac import giacsettings
sage: from sage.libs.giac import local_giacsettings
sage: gporig, gtorig = (giacsettings.proba_epsilon,giacsettings.threads)
sage: gp, gt = local_giacsettings(testf)(giacsettings.proba_epsilon,giacsettings.
↪threads)
sage: gporig == giacsettings.proba_epsilon
True
sage: gtorig == giacsettings.threads
True
sage: gp<gporig, gt-gtorig
(True, 2)
```

CHAPTER

FIVE

GMP-ECM

5.1 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 https://gitlab.inria.fr/zimmerma/ecm 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]
True
sage: result[1] in [3, 9, 27, 37, 111, 333, 999] or result[1]
True
sage: result = ecmfactor(999, 0.00, verbose=True)
Performing one curve with B1=0
Found factor in step 1: ...
sage: result[0]
True
sage: result[1] in [3, 9, 27, 37, 111, 333, 999] or result[1]
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 *␣
↪4357006235375344605345561005667974000505696611184208940783890278320995998159307781133050732832
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)
```

CHAPTER

SIX

GSL

6.1 GSL arrays

class sage.libs.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 SEVEN

LCALC

7.1 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: 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.compute_rank()
0
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
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: from sage.libs.lcalc.lcalc_Lfunction import *
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: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L = Lfunction_Zeta()
sage: L.find_zeros(10,29.1,.1)
[14.1347251417..., 21.0220396387..., 25.0108575801...]
```

find_zeros_via_N(*count=0*, *start=0*, *max_refine=1025*, *rank=-1*)

Find `count` zeros (in order of increasing magnitude) and output their imaginary parts. This function verifies that no zeros are missed, and that all values output are indeed zeros.

If this L-function is self-dual (if its Dirichlet coefficients are real, up to a tolerance of 1e-6), then only the zeros with positive imaginary parts are output. Their conjugates, which are also zeros, are not output.

INPUT:

- `count` – number of zeros to be found
- `start` – (default: 0) how many initial zeros to skip
- `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)

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: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi = DirichletGroup(5)[1]
sage: L = Lfunction_from_character(chi, type="complex")
sage: zeros = L.find_zeros_via_N(3)
sage: (zeros[0] - (-4.13290370521286)).abs() < 1e-8
True
sage: (zeros[1]  - 6.18357819545086).abs() < 1e-8
True
sage: (zeros[2]  - 8.45722917442320).abs() < 1e-8
True
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
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: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi = DirichletGroup(5)[2]  # Quadratic character
sage: L = Lfunction_from_character(chi, type="int")
sage: (L.hardy_z_function(0) - 0.231750947504).abs() < 1e-8
True
sage: L.hardy_z_function(0.5).imag().abs() < 1e-8
True
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi = DirichletGroup(5)[1]
sage: L = Lfunction_from_character(chi, type="complex")
sage: (L.hardy_z_function(0) - 0.793967590477).abs() < 1e-8
True
sage: L.hardy_z_function(0.5).imag().abs() < 1e-8
True
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: E = EllipticCurve([-82,0])
sage: L = Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: (L.hardy_z_function(2.1) - (-0.006431791768)).abs() < 1e-8
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: 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.value(0.5) - 0.231750947504016).abs() < 1e-8
True
sage: v = L.value(0.2 + 0.4*I)
sage: (v - (0.102558603193 + 0.190840777924*I)).abs() < 1e-8
True
sage: L = Lfunction_from_character(chi, type="double")
sage: (L.value(0.6) - 0.274633355856345).abs() < 1e-8
True
sage: v = L.value(0.6 + I)
sage: (v - (0.362258705721 + 0.43388825062*I)).abs() < 1e-8
True
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi = DirichletGroup(5)[1]
sage: L = Lfunction_from_character(chi, type="complex")
sage: v = L.value(0.5)
sage: (v - (0.763747880117 + 0.21696476751*I)).abs() < 1e-8
True
sage: v = L.value(0.6 + 5*I)
sage: (v - (0.702723260619 - 1.10178575243*I)).abs() < 1e-8
True
```

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L = Lfunction_Zeta()
sage: (L.value(0.5) + 1.46035450880).abs() < 1e-8
True
sage: v = L.value(0.4 + 0.5*I)
sage: (v - (-0.450728958517 - 0.780511403019*I)).abs() < 1e-8
True
```

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

Bases: *`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 arXiv 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

Note: 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: *`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 arXiv 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

Note: 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: *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 arXiv 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

Note: 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: *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-8
True
sage: (L.value(0.5, derivative=1) - 0.305999773835200).abs() < 1e-6
True
```

CHAPTER

EIGHT

LIBSINGULAR

8.1 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.

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: Singular library 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`.

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

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: `SageObject`

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

`ring()`

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

EXAMPLES:

```
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: *`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: *`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.

EXAMPLES:

```
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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: SageObject

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

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

Bases: *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: *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.

EXAMPLES:

```
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*)

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

EXAMPLES:

```
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.

EXAMPLES:

```
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*)

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

EXAMPLES:

```
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

EXAMPLES:

```
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.

EXAMPLES:

```
sage: from sage.libs.singular.function import list_of_functions
sage: 'groebner' in 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")
```

(continues on next page)

(continued from previous page)

```
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 (got 0 arguments, arity is CMD_12)
sage: factorize(f, 1, 2)
Traceback (most recent call last):
...
RuntimeError: error in Singular function call 'factorize':
Wrong number of arguments (got 3 arguments, arity is CMD_12)
sage: factorize(f, 1, 2, 3)
Traceback (most recent call last):
...
RuntimeError: error in Singular function call 'factorize':
Wrong number of arguments (got 4 arguments, arity is CMD_12)
```

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: Singular library 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")
```

(continues on next page)

(continued from previous page)

```
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) == matrix[0, 0] * matrix[1, 1] - matrix[0, 1] * matrix[1, 0]
True
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)
```

(continues on next page)

(continued from previous page)

```
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>
```

8.2 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.

8.3 libSingular: Conversion Routines and Initialisation

AUTHOR:

- Martin Albrecht <malb@informatik.uni-bremen.de>
- Miguel Marco <mmarco@unizar.es> (2021): added transcendental extensions over Q

`sage.libs.singular.singular.`**`get_resource`**(*id*)

Return a Singular "resource".

INPUT:

- `id` – a single-character string; see https://github.com/Singular/Singular/blob/spielwiese/resources/feResource.cc

OUTPUT:

A string, or `None`.

EXAMPLES:

```
sage: from sage.libs.singular.singular import get_resource
sage: get_resource('D')             # SINGULAR_DATA_DIR
'...'
sage: get_resource('i')             # SINGULAR_INFO_FILE
'.../singular...'
sage: get_resource('7') is None     # not defined
True
```

`sage.libs.singular.singular.`**`si2sa_resolution`**(*res*)

Pull the data from Singular resolution `res` to construct a Sage resolution.

INPUT:

- `res` – Singular resolution

The procedure is destructive and `res` is not usable afterward.

EXAMPLES:

```
sage: from sage.libs.singular.singular import si2sa_resolution
sage: from sage.libs.singular.function import singular_function
sage: module = singular_function("module")
sage: mres = singular_function('mres')

sage: S.<x,y,z,w> = PolynomialRing(QQ)
sage: I = S.ideal([y*w - z^2, -x*w + y*z, x*z - y^2])
sage: mod = module(I)
sage: r = mres(mod, 0)
sage: si2sa_resolution(r)
[
                                 [ y  x]
                                 [-z -y]
[z^2 - y*w y*z - x*w y^2 - x*z], [ w  z]
]
```

`sage.libs.singular.singular.`**`si2sa_resolution_graded`**(*res*, *degrees*)

Pull the data from Singular resolution `res` to construct a Sage resolution.

INPUT:

- `res` – Singular resolution
- `degrees` – list of integers or integer vectors

The procedure is destructive, and `res` is not usable afterward.

EXAMPLES:

```
sage: from sage.libs.singular.singular import si2sa_resolution_graded
sage: from sage.libs.singular.function import singular_function
sage: module = singular_function("module")
sage: mres = singular_function('mres')

sage: S.<x,y,z,w> = PolynomialRing(QQ)
sage: I = S.ideal([y*w - z^2, -x*w + y*z, x*z - y^2])
sage: mod = module(I)
sage: r = mres(mod, 0)
sage: res_mats, res_degs = si2sa_resolution_graded(r, (1, 1, 1, 1))
sage: res_mats
[
                                  [ y  x]
                                  [-z -y]
[z^2 - y*w y*z - x*w y^2 - x*z], [ w  z]
]
sage: res_degs
[[[2], [2], [2]], [[1, 1, 1], [1, 1, 1]]]
```

8.4 Wrapper for Singular's Polynomial Arithmetic

AUTHOR:

- Martin Albrecht (2009-07): refactoring

8.5 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: *`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_sb` 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 inhomogeneous 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 ordering 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.

EXAMPLES:

```
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.

EXAMPLES:

```
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()
```

(continues on next page)

(continued from previous page)

```
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.

EXAMPLES:

```
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*)

EXAMPLES:

```
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.

EXAMPLES:

```
sage: from sage.libs.singular.option import opt
sage: opt['deg_bound']
0
sage: opt['red_tail']
```

(continues on next page)

(continued from previous page)

```
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: *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.

EXAMPLES:

```
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

EXAMPLES:

```
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
```

8.6 Wrapper for Singular's Rings

AUTHORS:

- Martin Albrecht (2009-07): initial implementation
- Kwankyu Lee (2010-06): added matrix term order support
- Miguel Marco (2021): added transcendental extensions over Q

`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
<class 'sage.libs.singular.ring.ring_wrapper_Py'>
```

8.7 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: 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.

EXAMPLES:

```
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.

EXAMPLES:

```
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
```

ring()

Return the ring this strategy object is defined over.

EXAMPLES:

```
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: `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.

EXAMPLES:

```
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.

EXAMPLES:

```
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')
```

(continues on next page)

(continued from previous page)

```
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.

EXAMPLES:

```
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*)

EXAMPLES:

```
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*)

EXAMPLES:

```
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

NINE

GAP

9.1 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*)

Bases: `object`

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
```

9.2 Common global functions defined by GAP.

9.3 Long tests for GAP

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)
```

9.4 Utility functions for GAP

exception `sage.libs.gap.util.`**`GAPError`**

Bases: `ValueError`

Exceptions raised by the GAP library

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.**get_owned_objects**()

Helper to access the refcount dictionary from Python code

9.5 Library Interface to GAP

This module implements a fast C library interface to GAP. To use it, 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)
<class '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(_)
<class '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(_)
<class 'sage.rings.integer.Integer'>

sage: libgap.eval('5/3 + 7*E(3)').sage()                                       #␣
↪needs sage.rings.number_field
7*zeta3 + 5/3

sage: gens_of_group = libgap.AlternatingGroup(4).GeneratorsOfGroup()
sage: generators = gens_of_group.sage()
sage: generators   # a Sage list of Sage permutations!
[[2, 3, 1], [1, 3, 4, 2]]
sage: PermutationGroup(generators).cardinality()   # computed in Sage
12
```

(continues on next page)

(continued from previous page)

```
sage: libgap.AlternatingGroup(4).Size()            # computed in GAP
12
```

We can also specify which group in Sage the permutations should consider themselves as elements of when converted to Sage:

```
sage: A4 = groups.permutation.Alternating(4)
sage: generators = gens_of_group.sage(parent=A4); generators
[(1,2,3), (2,3,4)]
sage: all(gen.parent() is A4 for gen in generators)
True
```

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)
<class '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])
<class '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]
```

9.5.1 Using the GAP C library from Cython

Todo: Expand the following text

We are using the GAP API provided by the GAP project since GAP 4.10.

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.
- Dima Pasechnik (2018-09-18, GAP Days): started the port to native libgap API

class `sage.libs.gap.libgap.`**`Gap`**

Bases: `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 ] )
```

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 GAP.

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')
NULL
```

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
```

load_package (*pkg*)

If loading fails, raise a `RuntimeError` exception.

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')
NULL
```

set_seed(*seed=None*)

Reseed the standard GAP pseudo-random sources with the given seed.

Uses a random seed given by `current_randstate().ZZ_seed()` if `seed=None`. Otherwise the seed should be an integer.

EXAMPLES:

```
sage: libgap.set_seed(0)
0
sage: [libgap.Random(1, 10) for i in range(5)]
[2, 3, 3, 4, 2]
```

show()

Return statistics about the GAP owned object list

This includes the total memory allocated by GAP as returned by `libgap.eval('TotalMemoryAllocated()'), as well as garbage collection / object count statistics as returned by ``libgap.eval('GasmanStatistics')`, and finally the total number of GAP objects held by Sage as *GapElement* instances.

The value `livekb + deadkb` will roughly equal the total memory allocated for GAP objects (see `libgap.eval('TotalMemoryAllocated()')`).

Note: 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.collect()
sage: libgap.show()  # random output
{'gasman_stats': {'full': {'cumulative': 110,
   'deadbags': 321400,
   'deadkb': 12967,
   'freekb': 15492,
   'livebags': 396645,
   'livekb': 37730,
   'time': 110,
   'totalkb': 65536},
  'nfull': 1,
```

(continues on next page)

(continued from previous page)

```
  'npartial': 1},
 'nelements': 23123,
 'total_alloc': 3234234}
```

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')
NULL
```

zero()

Return (integer) zero in GAP.

OUTPUT:

A `GapElement`.

EXAMPLES:

```
sage: libgap.zero()
0
```

9.6 Short tests for GAP

`sage.libs.gap.test.`**test_write_to_file**()

Test that libgap can write to files

See Issue #16502, Issue #15833.

EXAMPLES:

```
sage: from sage.libs.gap.test import test_write_to_file
sage: test_write_to_file()
```

9.7 GAP element wrapper

This document describes the individual wrappers for various GAP elements. For general information about GAP, you should read the *libgap* module documentation.

class `sage.libs.gap.element.`**GapElement**

Bases: `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 *GAPError* exception is raised:

```
sage: libgap.eval('1/0')
Traceback (most recent call last):
...
GAPError: Error, Rational operations: <divisor> must not be zero
```

Also, a `GAPError` is raised if the input is not a simple expression:

```
sage: libgap.eval('1; 2; 3')
Traceback (most recent call last):
...
GAPError: can only evaluate a single statement
```

deepcopy(*mut*)

Return a deepcopy of this Gap object

Note that this is the same thing as calling `StructuralCopy` but much faster.

INPUT:

- `mut` – (boolean) whether to return a mutable copy

EXAMPLES:

```
sage: a = libgap([[0,1],[2,3]])
sage: b = a.deepcopy(1)
sage: b[0,0] = 5
sage: a
[ [ 0, 1 ], [ 2, 3 ] ]
sage: b
[ [ 5, 1 ], [ 2, 3 ] ]

sage: l = libgap([0,1])
sage: l.deepcopy(0).IsMutable()
false
sage: l.deepcopy(1).IsMutable()
true
```

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(_)
<class 'sage.rings.integer.Integer'>

sage: libgap(3/7).sage()
3/7
sage: type(_)
<class '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(_)
<... 'bool'>

sage: libgap('this is a string').sage()
'this is a string'
sage: type(_)
<... 'str'>

sage: x = libgap.Integers.Indeterminate("x")

sage: p = x^2 - 2*x + 3
sage: p.sage()
x^2 - 2*x + 3
sage: p.sage().parent()
Univariate Polynomial Ring in x over Integer Ring

sage: p = x^-2 + 3*x
sage: p.sage()
x^-2 + 3*x
sage: p.sage().parent()
```

(continues on next page)

(continued from previous page)

```
Univariate Laurent Polynomial Ring in x over Integer Ring

sage: p = (3 * x^2 + x) / (x^2 - 2)
sage: p.sage()
(3*x^2 + x)/(x^2 - 2)
sage: p.sage().parent()
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

class sage.libs.gap.element.**GapElement_Boolean**

Bases: *GapElement*

Derived class of GapElement for GAP boolean values.

EXAMPLES:

```
sage: b = libgap(True)
sage: type(b)
<class '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(_)
<class 'sage.libs.gap.element.GapElement_Boolean'>
sage: b.sage()
True
sage: 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: *GapElement*

Derived class of GapElement for GAP universal cyclotomics.

EXAMPLES:

```
sage: libgap.eval('E(3)')
E(3)
sage: type(_)
<class '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
```

class sage.libs.gap.element.**GapElement_FiniteField**

Bases: *GapElement*

Derived class of GapElement for GAP finite field elements.

EXAMPLES:

```
sage: libgap.eval('Z(5)^2')
Z(5)^2
sage: type(_)
<class '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(_)
<class '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:

A 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 ring is incompatible ...
```

class sage.libs.gap.element.**GapElement_Float**

Bases: *GapElement*

Derived class of GapElement for GAP floating point numbers.

EXAMPLES:

```
sage: i = libgap(123.5)
sage: type(i)
<class 'sage.libs.gap.element.GapElement_Float'>
sage: RDF(i)
123.5
sage: float(i)
123.5
```

sage (*ring=None*)

Return the Sage equivalent of the *GapElement_Float*

- ring – a floating point field or None (default). If not specified, the default Sage RDF is used.

OUTPUT:

A Sage double precision floating point number

EXAMPLES:

```
sage: a = libgap.eval("Float(3.25)").sage()
sage: a
3.25
sage: parent(a)
Real Double Field
```

class sage.libs.gap.element.**GapElement_Function**

Bases: *GapElement*

Derived class of GapElement for GAP functions.

EXAMPLES:

```
sage: f = libgap.Cycles
sage: type(f)
<class 'sage.libs.gap.element.GapElement_Function'>
```

class sage.libs.gap.element.**GapElement_Integer**

Bases: *GapElement*

Derived class of GapElement for GAP integers.

EXAMPLES:

```
sage: i = libgap(123)
sage: type(i)
<class '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)
<class 'sage.libs.gap.element.GapElement_Integer'>
sage: n.is_C_int()
True
sage: n.IsInt()
true

sage: N = libgap(2^130)
sage: type(N)
<class '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]
```

(continues on next page)

(continued from previous page)

```
sage: all( x in ZZ for x in _ )
True

sage: libgap(132).sage(ring=IntegerModRing(13))
2
sage: parent(_)
Ring of integers modulo 13
```

class sage.libs.gap.element.**GapElement_IntegerMod**

Bases: *GapElement*

Derived class of GapElement for GAP integers modulo an integer.

EXAMPLES:

```
sage: n = IntegerModRing(123)(13)
sage: i = libgap(n)
sage: type(i)
<class '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(_)
<class '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: *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)
<class '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(_)
<... '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)
<class 'sage.libs.gap.element.GapElement_List'>
```

(continues on next page)

(continued from previous page)

```
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 vector 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)
<class '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: *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
```

(continues on next page)

(continued from previous page)

```
<Gap function "Add">
sage: type(_)
<class 'sage.libs.gap.element.GapElement_MethodProxy'>
sage: lst.Add(1)
sage: lst
[ 1 ]
```

class sage.libs.gap.element.**GapElement_Permutation**

Bases: *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)
<class 'sage.libs.gap.element.GapElement_Permutation'>
```

sage(*parent=None*)

Return the Sage equivalent of the *GapElement*

If the permutation group is given as parent, this method is *much* faster.

EXAMPLES:

```
sage: perm_gap = libgap.eval('(1,5,2)(4,3,8)');  perm_gap
(1,5,2)(3,8,4)
sage: perm_gap.sage()
[5, 1, 8, 3, 2, 6, 7, 4]
sage: type(_)
<class 'sage.combinat.permutation.StandardPermutations_all_with_category.
↪element_class'>
sage: perm_gap.sage(PermutationGroup([(1,2),(1,2,3,4,5,6,7,8)]))
(1,5,2)(3,8,4)
sage: type(_)
<class 'sage.groups.perm_gps.permgroup_element.PermutationGroupElement'>
```

class sage.libs.gap.element.**GapElement_Rational**

Bases: *GapElement*

Derived class of GapElement for GAP rational numbers.

EXAMPLES:

```
sage: r = libgap(123/456)
sage: type(r)
<class '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(_)
<class 'sage.libs.gap.element.GapElement_Rational'>
sage: r.sage()
41/152
sage: type(_)
<class 'sage.rings.rational.Rational'>
```

class sage.libs.gap.element.**GapElement_Record**

Bases: *GapElement*

Derived class of GapElement for GAP records.

EXAMPLES:

```
sage: rec = libgap.eval('rec(a:=123, b:=456)')
sage: type(rec)
<class '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(_)
<... 'dict'>
```

Range checking is performed:

```
sage: rec['no_such_element']
Traceback (most recent call last):
...
GAPError: Error, Record Element: '<rec>.no_such_element' must have an assigned␣
↪value
```

record_name_to_index(*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(!)

EXAMPLES:

```
sage: rec = libgap.eval('rec(first:=123, second:=456)')
sage: rec.record_name_to_index('first')    # random output
1812
sage: rec.record_name_to_index('no_such_name') # random output
3776
```

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: sorted(rec)
[('a', 123), ('b', 456)]
sage: dict(rec)
{'a': 123, 'b': 456}
```

class sage.libs.gap.element.**GapElement_Ring**

Bases: *GapElement*

Derived class of GapElement for GAP rings (parents of ring elements).

EXAMPLES:

```
sage: i = libgap(ZZ)
sage: type(i)
<class '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_polynomial()

Construct a polynomial ring.

EXAMPLES:

```
sage: B = libgap(QQ['x'])
sage: B.ring_polynomial()
Univariate Polynomial Ring in x over Rational Field

sage: B = libgap(ZZ['x','y'])
sage: B.ring_polynomial()
Multivariate Polynomial Ring in x, y over Integer Ring
```

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

sage: libgap(QQ['x','y']).sage()
Multivariate Polynomial Ring in x, y over Rational Field
```

class sage.libs.gap.element.**GapElement_String**

Bases: *GapElement*

Derived class of GapElement for GAP strings.

EXAMPLES:

```
sage: s = libgap('string')
sage: type(s)
<class '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(_)
<class 'sage.libs.gap.element.GapElement_String'>
sage: str(s)
'string'
sage: s.sage()
'string'
sage: type(_)
<class 'str'>
```

9.8 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 (lib)gap

OUTPUT:

Float. Unix timestamp of the most recently changed GAP/LibGAP file(s). In particular, the timestamp increases whenever a gap package is added.

EXAMPLES:

```
sage: from sage.libs.gap.saved_workspace import timestamp
sage: timestamp()    # random output
1406642467.25684
sage: type(timestamp())
<... '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

TEN

LINBOX

10.1 Interface between flint matrices and linbox

This module only contains C++ code (and the interface is fully C compatible). It basically contains what used to be in the LinBox source code under interfaces/sage/linbox-sage.C written by M. Albrecht and C. Pernet. The functions available are:

- `void linbox_fmpz_mat_mul(fmpz_mat_t C, fmpz_mat_t A, fmpz_mat_t B)`: set `C` to be the result of the multiplication `A * B`
- `void linbox_fmpz_mat_charpoly(fmpz_poly_t cp, fmpz_mat_t A)`: set `cp` to be the characteristic polynomial of the square matrix `A`
- `void linbox_fmpz_mat_minpoly(fmpz_poly_t mp, fmpz_mat_t A)`: set `mp` to be the minimal polynomial of the square matrix `A`
- `size_t linbox_fmpz_mat_rank(fmpz_mat_t A)`: return the rank of the matrix `A`
- `void linbox_fmpz_mat_det(fmpz_t det, fmpz_mat_t A)`: set `det` to the determinant of the square matrix `A`

CHAPTER

ELEVEN

LRCALC

11.1 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://sites.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 $\mathfrak{gl}(3)$:

```
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 $\mathfrak{sl}(3)$ 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 $M' = ``maxrows`$ restricts the alphabet to $\{1, 2, \ldots, M\}$:

```
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=-1*)

Iterate over the skew LR tableaux of shape `outer / inner`.

INPUT:

- `outer` – a partition
- `inner` – a partition
- `weight` – a partition (optional)
- `maxrows` – a positive integer (optional)

OUTPUT: an iterator of `SkewTableau`

Specifying `maxrows` = M restricts the alphabet to $\{1, 2, \ldots, M\}$.

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: list(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 $S(\text{rank})$ 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=-1*)

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)]
```

CHAPTER

TWELVE

MPMATH

12.1 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)
```

(continues on next page)

(continued from previous page)

```
-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(_)
<class 'sage.rings.complex_mpfr.ComplexNumber'>
sage: a.call(a.polylog, 2, 1/2, parent=RDF)
0.5822405264650125
sage: type(_)
<class 'sage.rings.real_double...RealDoubleElement...'>
```

Check that Issue #11885 is fixed:

```
sage: a.call(a.ei, 1.0r, parent=float)
1.8951178163559366
```

Check that Issue #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')]
```

(continues on next page)

(continued from previous page)

```
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

THIRTEEN

NTL

13.1 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

FOURTEEN

PARI

14.1 Interface between Sage and PARI

14.1.1 Guide to real precision in the PARI interface

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

Internal representation and conversion between Sage and PARI

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

```
sage: x = 1.0
sage: x.precision()
53
sage: pari(x)
1.00000000000000
sage: pari(x).bitprecision()
64
```

With a higher precision:

```
sage: x = RealField(100).pi()
sage: x.precision()
100
sage: pari(x).bitprecision()
128
```

When converting back to Sage, the precision from PARI is taken:

```
sage: x = RealField(100).pi()
sage: y = pari(x).sage()
sage: y
3.1415926535897932384626433832793333156
sage: parent(y)
Real Field with 128 bits of precision
```

So `pari(x).sage()` is definitely not equal to `x` since it has 28 bogus bits.

Therefore, some care must be taken when juggling reals back and forth between Sage and PARI. The correct way of avoiding this is to convert `pari(x).sage()` 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). For instance, if we want to use the PARI library to compute `sqrt(pi)` with a precision of 100 bits:

```
sage: # needs sage.symbolic
sage: R = RealField(100)
sage: s = R(pi); s
3.1415926535897932384626433833
sage: p = pari(s).sqrt()
sage: x = p.sage(); 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
```

Output precision for printing

Even though PARI reals have a precision, not all significant bits are printed by default. The maximum number of digits when printing a PARI real can be set using the methods `Pari.set_real_precision_bits()` or `Pari.set_real_precision()`.

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

```
sage: pi = pari(RealField(1000).pi())
```

The default precision is 15 digits:

```
sage: pi
3.14159265358979
```

With a different precision:

```
sage: _ = pari.set_real_precision(50)
sage: pi
3.1415926535897932384626433832795028841971693993751
```

Back to the default:

```
sage: _ = pari.set_real_precision(15)
sage: pi
3.14159265358979
```

Input precision for function calls

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

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

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

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

In the second case, the precision can be given as the argument `precision` in the function call, with a default of 53 bits. The real precision set by `Pari.set_real_precision_bits()` or `Pari.set_real_precision()` is irrelevant.

In these examples, we convert to Sage to ensure that PARI's real precision is not used when printing the numbers. As explained before, this artificially increases the precision to a multiple of the wordsize.

```
sage: s = pari(1).sin(precision=180).sage(); print(s); print(parent(s))
0.841470984807896506652502321630298999622563060798371065673
Real Field with 192 bits of precision
sage: s = pari(1).sin(precision=40).sage(); print(s); print(parent(s))
0.841470984807896507
Real Field with 64 bits of precision
sage: s = pari(1).sin().sage(); print(s); print(parent(s))
0.841470984807896507
Real Field with 64 bits of precision
```

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

```
sage: pari(1.0).sin(precision=180).sage()
0.841470984807896507
sage: pari(1.0).sin(precision=40).sage()
0.841470984807896507
sage: pari(RealField(100).one()).sin().sage()
0.84147098480789650665250232163029899962
```

Elliptic curve functions

An elliptic curve given with exact a-invariants is considered an exact object. Therefore, you should set the precision for each method call individually:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: eta1 = e.elleta(precision=50)[0]
sage: eta1.sage()
3.6054636014326520859158205642077267748 # 64-bit
3.605463601432652085915820564           # 32-bit
sage: eta1 = e.elleta(precision=150)[0]
```

(continues on next page)

(continued from previous page)

```
sage: eta1.sage()
3.60546360143265208591582056420772677481026899659802474544438064142982049174O # 64-bit
3.60546360143265208591582056420772677481026899659802474544                    # 32-bit
```

14.2 Convert PARI objects to Sage types

`sage.libs.pari.convert_sage.`**`gen_to_sage`**(*z*, *locals=None*)

Convert a PARI gen to a Sage/Python object.

INPUT:

- `z` – PARI `gen`
- `locals` – optional dictionary used in fallback cases that involve `sage_eval()`

OUTPUT:

One of the following depending on the PARI type of `z`

- a `Integer` if `z` is an integer (type `t_INT`)
- a `Rational` if `z` is a rational (type `t_FRAC`)
- a `RealNumber` if `z` is a real number (type `t_REAL`). The precision will be equivalent.
- a `NumberFieldElement_quadratic` or a `ComplexNumber` if `z` is a complex number (type `t_COMPLEX`). The former is used when the real and imaginary parts are integers or rationals and the latter when they are floating point numbers. In that case The precision will be the maximal precision of the real and imaginary parts.
- a Python list if `z` is a vector or a list (type `t_VEC`, `t_COL`)
- a Python string if `z` is a string (type `t_STR`)
- a Python list of Python integers if `z` is a small vector (type `t_VECSMALL`)
- a matrix if `z` is a matrix (type `t_MAT`)
- a padic element (type `t_PADIC`)
- a `Infinity` if `z` is an infinity (type `t_INF`)

EXAMPLES:

```
sage: from sage.libs.pari.convert_sage import gen_to_sage
```

Converting an integer:

```
sage: z = pari('12'); z
12
sage: z.type()
't_INT'
sage: a = gen_to_sage(z); a
12
sage: a.parent()
Integer Ring

sage: gen_to_sage(pari('7^42'))
311973482284542371301330321821976049
```

Converting a rational number:

```
sage: z = pari('389/17'); z
389/17
sage: z.type()
't_FRAC'
sage: a = gen_to_sage(z); a
389/17
sage: a.parent()
Rational Field

sage: gen_to_sage(pari('5^30 / 3^50'))
931322574615478515625/717897987691852588770249
```

Converting a real number:

```
sage: pari.set_real_precision(70)
15
sage: z = pari('1.234'); z
1.234000000000000000000000000000000000000000000000000000000000000000000
sage: a = gen_to_sage(z); a                                                   #␣
↪needs sage.rings.real_mpfr
1.234000000000000000000000000000000000000000000000000000000000000000000000000
sage: a.parent()                                                              #␣
↪needs sage.rings.real_mpfr
Real Field with 256 bits of precision
sage: pari.set_real_precision(15)
70
sage: a = gen_to_sage(pari('1.234')); a                                       #␣
↪needs sage.rings.real_mpfr
1.23400000000000000
sage: a.parent()                                                              #␣
↪needs sage.rings.real_mpfr
Real Field with 64 bits of precision
```

For complex numbers, the parent depends on the PARI type:

```
sage: z = pari('(3+I)'); z
3 + I
sage: z.type()
't_COMPLEX'
sage: a = gen_to_sage(z); a                                                   #␣
↪needs sage.rings.number_field
i + 3
sage: a.parent()                                                              #␣
↪needs sage.rings.number_field
Number Field in i with defining polynomial x^2 + 1 with i = 1*I

sage: z = pari('(3+I)/2'); z
3/2 + 1/2*I
sage: a = gen_to_sage(z); a                                                   #␣
↪needs sage.rings.number_field
1/2*i + 3/2
sage: a.parent()                                                              #␣
↪needs sage.rings.number_field
Number Field in i with defining polynomial x^2 + 1 with i = 1*I

sage: z = pari('1.0 + 2.0*I'); z
```

(continues on next page)

(continued from previous page)

```
1.00000000000000 + 2.00000000000000*I
sage: a = gen_to_sage(z); a                                                  #␣
↪needs sage.rings.real_mpfr
1.0000000000000000000 + 2.0000000000000000000*I
sage: a.parent()                                                             #␣
↪needs sage.rings.real_mpfr
Complex Field with 64 bits of precision

sage: z = pari('1 + 1.0*I'); z
1 + 1.00000000000000*I
sage: a = gen_to_sage(z); a                                                  #␣
↪needs sage.rings.real_mpfr
1.0000000000000000000 + 1.0000000000000000000*I
sage: a.parent()                                                             #␣
↪needs sage.rings.real_mpfr
Complex Field with 64 bits of precision

sage: z = pari('1.0 + 1*I'); z
1.00000000000000 + I
sage: a = gen_to_sage(z); a                                                  #␣
↪needs sage.rings.real_mpfr
1.0000000000000000000 + 1.0000000000000000000*I
sage: a.parent()                                                             #␣
↪needs sage.rings.real_mpfr
Complex Field with 64 bits of precision
```

Converting polynomials:

```
sage: f = pari('(2/3)*x^3 + x - 5/7 + y')
sage: f.type()
't_POL'

sage: R.<x,y> = QQ[]
sage: gen_to_sage(f, {'x': x, 'y': y})
2/3*x^3 + x + y - 5/7
sage: parent(gen_to_sage(f, {'x': x, 'y': y}))
Multivariate Polynomial Ring in x, y over Rational Field

sage: # needs sage.symbolic
sage: x,y = SR.var('x,y')
sage: gen_to_sage(f, {'x': x, 'y': y})
2/3*x^3 + x + y - 5/7
sage: parent(gen_to_sage(f, {'x': x, 'y': y}))
Symbolic Ring

sage: gen_to_sage(f)
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Converting vectors:

```
sage: # needs sage.rings.number_field sage.rings.real_mpfr
sage: z1 = pari('[-3, 2.1, 1+I]'); z1
[-3, 2.10000000000000, 1 + I]
sage: z2 = pari('[1.0*I, [1,2]]~'); z2
[1.00000000000000*I, [1, 2]]~
```

(continues on next page)

(continued from previous page)

```
sage: z1.type(), z2.type()
('t_VEC', 't_COL')
sage: a1 = gen_to_sage(z1)
sage: a2 = gen_to_sage(z2)
sage: type(a1), type(a2)
(<... 'list'>, <... 'list'>)
sage: [parent(b) for b in a1]
[Integer Ring,
 Real Field with 64 bits of precision,
 Number Field in i with defining polynomial x^2 + 1 with i = 1*I]
sage: [parent(b) for b in a2]
[Complex Field with 64 bits of precision, <... 'list'>]

sage: z = pari('Vecsmall([1,2,3,4])')
sage: z.type()
't_VECSMALL'
sage: a = gen_to_sage(z); a
[1, 2, 3, 4]
sage: type(a)
<... 'list'>
sage: [parent(b) for b in a]
[<... 'int'>, <... 'int'>, <... 'int'>, <... 'int'>]
```

Matrices:

```
sage: z = pari('[1,2;3,4]')
sage: z.type()
't_MAT'

sage: # needs sage.modules
sage: a = gen_to_sage(z); a
[1 2]
[3 4]
sage: a.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

Conversion of p-adics:

```
sage: # needs sage.rings.padics
sage: z = pari('569 + O(7^8)'); z
2 + 4*7 + 4*7^2 + 7^3 + O(7^8)
sage: a = gen_to_sage(z); a
2 + 4*7 + 4*7^2 + 7^3 + O(7^8)
sage: a.parent()
7-adic Field with capped relative precision 8
```

Conversion of infinities:

```
sage: gen_to_sage(pari('oo'))
+Infinity
sage: gen_to_sage(pari('-oo'))
-Infinity
```

Conversion of strings:

```
sage: s = pari('"foo"').sage(); s
'foo'
```

(continues on next page)

(continued from previous page)

```
sage: type(s)
<class 'str'>
```

`sage.libs.pari.convert_sage.`**`new_gen_from_integer`**(*self*)

`sage.libs.pari.convert_sage.`**`new_gen_from_rational`**(*self*)

`sage.libs.pari.convert_sage.`**`pari_divisors_small`**(*self*)

Return the list of divisors of this number using PARI `divisorsu`.

See also:

This method is better used through `sage.rings.integer.Integer.divisors()`.

EXAMPLES:

```
sage: from sage.libs.pari.convert_sage import pari_divisors_small
sage: pari_divisors_small(4)
[1, 2, 4]
```

The integer must fit into an unsigned long:

```
sage: pari_divisors_small(-4)
Traceback (most recent call last):
...
AssertionError
sage: pari_divisors_small(2**65)
Traceback (most recent call last):
...
AssertionError
```

`sage.libs.pari.convert_sage.`**`pari_is_prime`**(*p*)

Return whether `p` is a prime.

The caller must ensure that `p.value` fits in a long.

EXAMPLES:

```
sage: from sage.libs.pari.convert_sage import pari_is_prime
sage: pari_is_prime(2)
True
sage: pari_is_prime(3)
True
sage: pari_is_prime(1)
False
sage: pari_is_prime(4)
False
```

Its recommended to use `sage.rings.integer.Integer.is_prime()`, which checks overflow. The following is incorrect, because the number does not fit into a long:

```
sage: pari_is_prime(2**64 + 2)
True
```

`sage.libs.pari.convert_sage.`**`pari_is_prime_power`**(*q, get_data*)

Return whether `q` is a prime power.

The caller must ensure that `q.value` fits in a long.

OUTPUT:

If `get_data` return a tuple of the prime and the exponent. Otherwise return a boolean.

EXAMPLES:

```
sage: from sage.libs.pari.convert_sage import pari_is_prime_power
sage: pari_is_prime_power(2, False)
True
sage: pari_is_prime_power(2, True)
(2, 1)
sage: pari_is_prime_power(4, False)
True
sage: pari_is_prime_power(4, True)
(2, 2)
sage: pari_is_prime_power(6, False)
False
sage: pari_is_prime_power(6, True)
(6, 0)
```

Its recommended to use `sage.rings.integer.Integer.is_prime_power()`, which checks overflow. The following is incorrect, because the number does not fit into a long:

```
sage: pari_is_prime_power(2**64 + 2, False)
True
```

`sage.libs.pari.convert_sage.`**`pari_maxprime`**`()`

Return to which limit PARI has computed the primes.

EXAMPLES:

```
sage: from sage.libs.pari.convert_sage import pari_maxprime
sage: a = pari_maxprime()
sage: res = prime_range(2, 2*a)
sage: b = pari_maxprime()
sage: b >= 2*a
True
```

`sage.libs.pari.convert_sage.`**`pari_prime_range`**`(`*c_start*, *c_stop*, *py_ints=False*`)`

Return a list of all primes between `start` and `stop - 1`, inclusive.

See also:

`sage.rings.fast_arith.prime_range()`

`sage.libs.pari.convert_sage.`**`set_integer_from_gen`**`(`*self*, *x*`)`

EXAMPLES:

```
sage: [Integer(pari(x)) for x in [1, 2^60, 2., GF(3)(1), GF(9,'a')(2)]]          #␣
↪needs sage.rings.finite_rings
[1, 1152921504606846976, 2, 1, 2]
sage: Integer(pari(2.1)) # indirect doctest
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral real number to an Integer
```

`sage.libs.pari.convert_sage.`**`set_rational_from_gen`**`(`*self*, *x*`)`

EXAMPLES:

```
sage: [Rational(pari(x)) for x in [1, 1/2, 2^60, 2., GF(3)(1), GF(9,'a')(2)]]    #␣
↪needs sage.rings.finite_rings
[1, 1/2, 1152921504606846976, 2, 1, 2]
sage: Rational(pari(2.1)) # indirect doctest
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral real number to an Integer
```

14.3 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: `RingElement`

Element of Pari pseudo-ring.

class sage.rings.pari_ring.**PariRing**

Bases: `Singleton`, `Parent`

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.

EXAMPLES:

```
sage: R = PariRing()
sage: R.random_element().parent() is R
True
sage: R(5) <= R.random_element(5,13) < R(13)
True
sage: R.random_element(distribution="1/n").parent() is R
True
```

zeta()

Return -1.

EXAMPLES:

```
sage: R = PariRing()
sage: R.zeta()
-1
```

CHAPTER FIFTEEN

SYMMETRICA

15.1 Symmetrica library

sage.libs.symmetrica.symmetrica.**bdg_symmetrica**(*part*, *perm*)

Calculates the irreducible 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
```

(continues on next page)

(continued from previous page)

```
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 homogeneous(=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.**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]
```

(continues on next page)

(continued from previous page)

```
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 irreducible 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, labeled 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 irreducible 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*)

Return 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*)

Return 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 . It uses the algorithm of Nijenhuis and 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 irreducible 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

SIXTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX