Sage Reference Manual: Numerical Optimization

Release 7.0

The Sage Development Team

January 20, 2016

CONTENTS

CHAPTER

ONE

KNAPSACK PROBLEMS

This module implements a number of solutions to various knapsack problems, otherwise known as linear integer programming problems. Solutions to the following knapsack problems are implemented:

- Solving the subset sum problem for super-increasing sequences.
- General case using Linear Programming

AUTHORS:

- Minh Van Nguyen (2009-04): initial version
- Nathann Cohen (2009-08): Linear Programming version

1.1 Definition of Knapsack problems

You have already had a knapsack problem, so you should know, but in case you do not, a knapsack problem is what happens when you have hundred of items to put into a bag which is too small, and you want to pack the most useful of them.

When you formally write it, here is your problem:

- Your bag can contain a weight of at most W.
- Each item i has a weight w_i.
- Each item i has a usefulness u_i.

You then want to maximize the total usefulness of the items you will store into your bag, while keeping sure the weight of the bag will not go over W.

As a linear program, this problem can be represented this way (if you define b_i as the binary variable indicating whether the item i is to be included in your bag):

$$\begin{aligned} \text{Maximize: } & \sum_i b_i u_i \\ \text{Such that: } & \sum_i b_i w_i \leq W \\ & \forall i, b_i \text{ binary variable} \end{aligned}$$

(For more information, see the Wikipedia article Knapsack_problem)

1.2 Examples

If your knapsack problem is composed of three items (weight, value) defined by (1,2), (1.5,1), (0.5,3), and a bag of maximum weight 2, you can easily solve it this way:

```
sage: from sage.numerical.knapsack import knapsack
sage: knapsack( [(1,2), (1.5,1), (0.5,3)], max=2)
[5.0, [(1, 2), (0.500000000000000, 3)]]
```

1.3 Super-increasing sequences

We can test for whether or not a sequence is super-increasing:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: seq = Superincreasing(L)
sage: seq
Super-increasing sequence of length 8
sage: seq.is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
False
```

Solving the subset sum problem for a super-increasing sequence and target sum:

```
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).subset_sum(98)
[69, 21, 5, 2, 1]
```

class `sage.numerical.knapsack.`**`Superincreasing`**`(`*seq=None*`)`

Bases: `sage.structure.sage_object.SageObject`

A class for super-increasing sequences.

Let $L = (a_1, a_2, a_3, \dots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has only one element, it is also defined to be a super-increasing sequence.

If `seq` is `None`, then construct an empty sequence. By definition, this empty sequence is not super-increasing.

INPUT:

- `seq` – (default: `None`) a non-empty sequence.

EXAMPLES:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
```

```
False
sage: seq = Superincreasing(); seq
An empty sequence.
sage: seq = Superincreasing([1, 3, 6]); seq
Super-increasing sequence of length 3
sage: seq = Superincreasing(list([1, 2, 5, 21, 69, 189, 376, 919])); seq
Super-increasing sequence of length 8
```

is_superincreasing(*seq=None*)

Determine whether or not `seq` is super-increasing.

If `seq=None` then determine whether or not `self` is super-increasing.

Let $L = (a_1, a_2, a_3, \dots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has exactly one element, then it is also defined to be a super-increasing sequence.

INPUT:

- `seq` – (default: `None`) a sequence to test

OUTPUT:

- If `seq` is `None`, then test `self` to determine whether or not it is super-increasing. In that case, return `True` if `self` is super-increasing; `False` otherwise.
- If `seq` is not `None`, then test `seq` to determine whether or not it is super-increasing. Return `True` if `seq` is super-increasing; `False` otherwise.

EXAMPLES:

By definition, an empty sequence is not super-increasing:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: Superincreasing().is_superincreasing([])
False
sage: Superincreasing().is_superincreasing()
False
sage: Superincreasing().is_superincreasing(tuple())
False
sage: Superincreasing().is_superincreasing(())
False
```

But here is an example of a super-increasing sequence:

```
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: L = (1, 2, 5, 21, 69, 189, 376, 919)
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can have zero as one of its elements:

```
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can be of length 1:

```
sage: Superincreasing([randint(0, 100)]).is_superincreasing()
True
```

TESTS:

The sequence must contain only integers:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1.0, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
Traceback (most recent call last):
...
TypeError: Element e (= 1.00000000000000) of seq must be a non-negative integer.
sage: L = [1, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
Traceback (most recent call last):
...
TypeError: Element e (= 2.10000000000000) of seq must be a non-negative integer.
```

largest_less_than(*N*)

Return the largest integer in the sequence `self` that is less than or equal to `N`.

This function narrows down the candidate solution using a binary trim, similar to the way binary search halves the sequence at each iteration.

INPUT:

- `N` – integer; the target value to search for.

OUTPUT:

The largest integer in `self` that is less than or equal to `N`. If no solution exists, then return `None`.

EXAMPLES:

When a solution is found, return it:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(207)
179
sage: L = (2, 3, 7, 25, 67, 179, 356, 819)
sage: Superincreasing(L).largest_less_than(2)
2
```

But if no solution exists, return `None`:

```
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(-1) is None
True
```

TESTS:

The target `N` must be an integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
```

```
sage: Superincreasing(L).largest_less_than(2.30)
Traceback (most recent call last):
...
TypeError: N (= 2.30000000000000) must be an integer.
```

The sequence that `self` represents must also be non-empty:

```
sage: Superincreasing([]).largest_less_than(2)
Traceback (most recent call last):
...
ValueError: seq must be a super-increasing sequence
sage: Superincreasing(list()).largest_less_than(2)
Traceback (most recent call last):
...
ValueError: seq must be a super-increasing sequence
```

subset_sum(N)

Solving the subset sum problem for a super-increasing sequence.

Let $S = (s_1, s_2, s_3, \dots, s_n)$ be a non-empty sequence of non-negative integers, and let $N \in \mathbf{Z}$ be non-negative. The subset sum problem asks for a subset $A \subseteq S$ all of whose elements sum to N. This method specializes the subset sum problem to the case of super-increasing sequences. If a solution exists, then it is also a super-increasing sequence.

Note: This method only solves the subset sum problem for super-increasing sequences. In general, solving the subset sum problem for an arbitrary sequence is known to be computationally hard.

INPUT:

- `N` – a non-negative integer.

OUTPUT:

- A non-empty subset of `self` whose elements sum to `N`. This subset is also a super-increasing sequence. If no such subset exists, then return the empty list.

ALGORITHMS:

The algorithm used is adapted from page 355 of [HPS08].

EXAMPLES:

Solving the subset sum problem for a super-increasing sequence and target sum:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).subset_sum(98)
[69, 21, 5, 2, 1]
```

TESTS:

The target `N` must be a non-negative integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).subset_sum(-6)
Traceback (most recent call last):
...
TypeError: N (= -6) must be a non-negative integer.
sage: Superincreasing(L).subset_sum(-6.2)
Traceback (most recent call last):
```

```
...
TypeError: N (= -6.20000000000000) must be a non-negative integer.
```

The sequence that `self` represents must only contain non-negative integers:

```
sage: L = [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1]
sage: Superincreasing(L).subset_sum(1)
Traceback (most recent call last):
...
TypeError: Element e (= -10) of seq must be a non-negative integer.
```

REFERENCES:

`sage.numerical.knapsack.`**`knapsack`**(*seq*, *binary=True*, *max=1*, *value_only=False*, *solver=None*, *verbose=0*)

Solves the knapsack problem

For more information on the knapsack problem, see the documentation of the `knapsack module` or the Wikipedia article Knapsack_problem.

INPUT:

- `seq` – Two different possible types:
  - A sequence of tuples `(weight, value, something1, something2, ...)`. Note that only the first two coordinates (`weight` and `values`) will be taken into account. The rest (if any) will be ignored. This can be useful if you need to attach some information to the items.
  - A sequence of reals (a value of 1 is assumed).
- `binary` – When set to `True`, an item can be taken 0 or 1 time. When set to `False`, an item can be taken any amount of times (while staying integer and positive).
- `max` – Maximum admissible weight.
- `value_only` – When set to `True`, only the maximum useful value is returned. When set to `False`, both the maximum useful value and an assignment are returned.
- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the documentation of class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

If `value_only` is set to `True`, only the maximum useful value is returned. Else (the default), the function returns a pair `[value,list]`, where `list` can be of two types according to the type of `seq`:

- The list of tuples $(w_i, u_i, ...)$ occurring in the solution.
- A list of reals where each real is repeated the number of times it is taken into the solution.

EXAMPLES:

If your knapsack problem is composed of three items `(weight, value)` defined by `(1,2), (1.5,1), (0.5,3)`, and a bag of maximum weight 2, you can easily solve it this way:

```
sage: from sage.numerical.knapsack import knapsack
sage: knapsack( [(1,2), (1.5,1), (0.5,3)], max=2)
[5.0, [(1, 2), (0.500000000000000, 3)]]

sage: knapsack( [(1,2), (1.5,1), (0.5,3)], max=2, value_only=True)
5.0
```

Besides weight and value, you may attach any data to the items:

```
sage: from sage.numerical.knapsack import knapsack
sage: knapsack( [(1, 2, 'spam'), (0.5, 3, 'a', 'lot')])
[3.0, [(0.500000000000000, 3, 'a', 'lot')]]
```

In the case where all the values (usefulness) of the items are equal to one, you do not need embarrass yourself with the second values, and you can just type for items $(1,1), (1.5,1), (0.5,1)$ the command:

```
sage: from sage.numerical.knapsack import knapsack
sage: knapsack([1,1.5,0.5], max=2, value_only=True)
2.0
```

CHAPTER

TWO

MIXED INTEGER LINEAR PROGRAMMING

This module implements classes and methods for the efficient solving of Linear Programs (LP) and Mixed Integer Linear Programs (MILP).

Do you want to understand how the simplex method works? See the `interactive_simplex_method` module (educational purposes only)

2.1 Definition

A linear program (LP) is an optimization problem in the following form

$$\max\{c^T x \mid Ax \leq b, x \geq 0\}$$

with given $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ and unknown $x \in \mathbb{R}^n$. If some or all variables in the vector x are restricted over the integers $\mathbb{Z}$, the problem is called mixed integer linear program (MILP). A wide variety of problems in optimization can be formulated in this standard form. Then, solvers are able to calculate a solution.

2.2 Example

Imagine you want to solve the following linear system of three equations:

- $w_0 + w_1 + w_2 - 14w_3 = 0$
- $w_1 + 2w_2 - 8w_3 = 0$
- $2w_2 - 3w_3 = 0$

and this additional inequality:

- $w_0 - w_1 - w_2 \geq 0$

where all $w_i \in \mathbb{Z}^+$. You know that the trivial solution is $w_i = 0$, but what is the first non-trivial one with $w_3 \geq 1$?

A mixed integer linear program can give you an answer:

1. You have to create an instance of `MixedIntegerLinearProgram` and – in our case – specify that it is a minimization.
2. Create a dictionary `w` of integer variables `w` via `w = p.new_variable(integer=True)` (note that **by default all variables are non-negative**, cf `new_variable()`).
3. Add those three equations as equality constraints via `add_constraint`.
4. Also add the inequality constraint.
5. Add an inequality constraint $w_3 \geq 1$ to exclude the trivial solution.

6. By default, all variables are non-negative. We remove that constraint via `p.set_min(variable, None)`, see `set_min`.
7. Specify the objective function via `set_objective`. In our case that is just w_3. If it is a pure constraint satisfaction problem, specify it as `None`.
8. To check if everything is set up correctly, you can print the problem via `show`.
9. `Solve` it and print the solution.

The following example shows all these steps:

```
sage: p = MixedIntegerLinearProgram(maximization=False, solver = "GLPK")
sage: w = p.new_variable(integer=True, nonnegative=True)
sage: p.add_constraint(w[0] + w[1] + w[2] - 14*w[3] == 0)
sage: p.add_constraint(w[1] + 2*w[2] - 8*w[3] == 0)
sage: p.add_constraint(2*w[2] - 3*w[3] == 0)
sage: p.add_constraint(w[0] - w[1] - w[2] >= 0)
sage: p.add_constraint(w[3] >= 1)
sage: _ = [ p.set_min(w[i], None) for i in range(1,4) ]
sage: p.set_objective(w[3])
sage: p.show()
Minimization:
   x_3
Constraints:
  0.0 <= x_0 + x_1 + x_2 - 14.0 x_3 <= 0.0
  0.0 <= x_1 + 2.0 x_2 - 8.0 x_3 <= 0.0
  0.0 <= 2.0 x_2 - 3.0 x_3 <= 0.0
  - x_0 + x_1 + x_2 <= 0.0
  - x_3 <= -1.0
Variables:
  x_0 is an integer variable (min=0.0, max=+oo)
  x_1 is an integer variable (min=-oo, max=+oo)
  x_2 is an integer variable (min=-oo, max=+oo)
  x_3 is an integer variable (min=-oo, max=+oo)
sage: print 'Objective Value:', p.solve()
Objective Value: 2.0
sage: for i, v in p.get_values(w).iteritems():
....:     print 'w_%s = %s' % (i, int(round(v)))
w_0 = 15
w_1 = 10
w_2 = 3
w_3 = 2
```

Different backends compute with different base fields, for example:

```
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: p.base_ring()
Real Double Field
sage: x = p.new_variable(real=True, nonnegative=True)
sage: 0.5 + 3/2*x[1]
0.5 + 1.5*x_0

sage: p = MixedIntegerLinearProgram(solver='ppl')
sage: p.base_ring()
Rational Field
sage: x = p.new_variable(nonnegative=True)
sage: 0.5 + 3/2*x[1]
1/2 + 3/2*x_0
```

2.3 Linear Variables and Expressions

The underlying linear programming backends always work with matrices where each column corresponds to a linear variable. These variables can be accessed using the `MixedIntegerLinearProgram.gen()` method or by calling with a dictionary variable index to coefficient:

```
sage: mip = MixedIntegerLinearProgram()
sage: 5 + mip.gen(0) + 2*mip.gen(1)
5 + x_0 + 2*x_1
sage: mip({-1:5, 0:1, 1:2})
5 + x_0 + 2*x_1
```

However, this alone is often not convenient to construct a linear program. To make your code more readable, you can construct `MIPVariable` objects that can be arbitrarily named and indexed. Internally, this is then translated back to the x_i variables. For example:

```
sage: mip.<a,b> = MixedIntegerLinearProgram()
sage: a
MIPVariable of dimension 1
sage: 5 + a[1] + 2*b[3]
5 + x_0 + 2*x_1
```

Indices can be any object, not necessarily integers. Multi-indices are also allowed:

```
sage: a[4, 'string', QQ]
x_2
sage: a[4, 'string', QQ] - 7*b[2]
x_2 - 7*x_3
sage: mip.show()
Maximization:

Constraints:
Variables:
  a[1] = x_0 is a continuous variable (min=-oo, max=+oo)
  b[3] = x_1 is a continuous variable (min=-oo, max=+oo)
  a[(4, 'string', Rational Field)] = x_2 is a continuous variable (min=-oo, max=+oo)
  b[2] = x_3 is a continuous variable (min=-oo, max=+oo)
```

2.4 Index of functions and methods

Below are listed the methods of `MixedIntegerLinearProgram`. This module also implements the `MIPSolverException` exception, as well as the `MIPVariable` class.

`add_constraint()`	Adds a constraint to the `MixedIntegerLinearProgram`
`base_ring()`	Return the base ring
`best_known_objective_bound()`	Return the value of the currently best known bound
`constraints()`	Returns a list of constraints, as 3-tuples
`get_backend()`	Returns the backend instance used
`get_max()`	Returns the maximum value of a variable
`get_min()`	Returns the minimum value of a variable
`get_objective_value()`	Return the value of the objective function
`get_relative_objective_gap()`	Return the relative objective gap of the best known solution
`get_values()`	Return values found by the previous call to `solve()`

Continued on next page

Table 2.1 – continued from previous page

is_binary()	Tests whether the variable e is binary
is_integer()	Tests whether the variable is an integer
is_real()	Tests whether the variable is real
linear_constraints_parent()	Return the parent for all linear constraints
linear_function()	Construct a new linear function
linear_functions_parent()	Return the parent for all linear functions
new_variable()	Returns an instance of MIPVariable associated
number_of_constraints()	Returns the number of constraints assigned so far
number_of_variables()	Returns the number of variables used so far
polyhedron()	Returns the polyhedron defined by the Linear Program
remove_constraint()	Removes a constraint from self
remove_constraints()	Remove several constraints
set_binary()	Sets a variable or a MIPVariable as binary
set_integer()	Sets a variable or a MIPVariable as integer
set_max()	Sets the maximum value of a variable
set_min()	Sets the minimum value of a variable
set_objective()	Sets the objective of the MixedIntegerLinearProgram
set_problem_name()	Sets the name of the MixedIntegerLinearProgram
set_real()	Sets a variable or a MIPVariable as real
show()	Displays the MixedIntegerLinearProgram in a human-readable
solve()	Solves the MixedIntegerLinearProgram
solver_parameter()	Return or define a solver parameter
sum()	Efficiently computes the sum of a sequence of LinearFunction elements
write_lp()	Write the linear program as a LP file
write_mps()	Write the linear program as a MPS file

AUTHORS:

- Risan (2012/02): added extension for exact computation

exception `sage.numerical.mip.`**`MIPSolverException`**`(`*value*`)`

Bases: `exceptions.RuntimeError`

Exception raised when the solver fails.

class `sage.numerical.mip.`**`MIPVariable`**

Bases: `sage.structure.element.Element`

`MIPVariable` is a variable used by the class `MixedIntegerLinearProgram`.

> **Warning:** You should not instantiate this class directly. Instead, use `MixedIntegerLinearProgram.new_variable()`.

`items``()`

Returns the pairs (keys,value) contained in the dictionary.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[0] + v[1])
sage: v.items()
[(0, x_0), (1, x_1)]
```

`keys``()`

Returns the keys already defined in the dictionary.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[0] + v[1])
sage: v.keys()
[0, 1]
```

set_max(*max*)

Sets an upper bound on the variable.

INPUT:

- `max` – an upper bound, or `None` to mean that the variable is unbounded.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(real=True, nonnegative=True)
sage: p.get_max(v)
sage: p.get_max(v[0])
sage: p.set_max(v,4)
sage: p.get_max(v)
4
sage: p.get_max(v[0])
4.0
```

set_min(*min*)

Sets a lower bound on the variable.

INPUT:

- `min` – a lower bound, or `None` to mean that the variable is unbounded.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(real=True, nonnegative=True)
sage: p.get_min(v)
0
sage: p.get_min(v[0])
0.0
sage: p.set_min(v,4)
sage: p.get_min(v)
4
sage: p.get_min(v[0])
4.0
```

values()

Returns the symbolic variables associated to the current dictionary.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[0] + v[1])
sage: v.values()
[x_0, x_1]
```

class `sage.numerical.mip.`**MIPVariableParent**

Bases: `sage.structure.parent.Parent`

Parent for `MIPVariable`.

Warning: This class is for internal use. You should not instantiate it yourself. Use `MixedIntegerLinearProgram.new_variable()` to generate mip variables.

Element
alias of `MIPVariable`

class `sage.numerical.mip.`**`MixedIntegerLinearProgram`**
Bases: `sage.structure.sage_object.SageObject`

The `MixedIntegerLinearProgram` class is the link between Sage, linear programming (LP) and mixed integer programming (MIP) solvers.

A Mixed Integer Linear Program (MILP) consists of variables, linear constraints on these variables, and an objective function which is to be maximised or minimised under these constraints.

See the Wikipedia article Linear_programming for further information on linear programming, and the `MILP module` for its use in Sage.

INPUT:

- `solver` – selects a solver:
  - GLPK (`solver="GLPK"`). See the GLPK web site.
  - COIN Branch and Cut (`solver="Coin"`). See the COIN-OR web site.
  - CPLEX (`solver="CPLEX"`). See the CPLEX web site.
  - **Gurobi (`solver="Gurobi"`). See the Gurobi** web site.
  - **CVXOPT (`solver="CVXOPT"`). See the CVXOPT** web site.
  - **PPL (`solver="PPL"`). See the PPL** web site.
  - If `solver=None` (default), the default solver is used (see `default_mip_solver()`)
- `maximization`
  - When set to `True` (default), the `MixedIntegerLinearProgram` is defined as a maximization.
  - When set to `False`, the `MixedIntegerLinearProgram` is defined as a minimization.
- `constraint_generation` – Only used when `solver=None`.
  - When set to `True`, after solving the `MixedIntegerLinearProgram`, it is possible to add a constraint, and then solve it again. The effect is that solvers that do not support this feature will not be used.
  - Defaults to `False`.

Warning: All LP variables are non-negative by default (see `new_variable()` and `set_min()`).

See also:

- `default_mip_solver()` – Returns/Sets the default MIP solver.

EXAMPLES:

Computation of a maximum stable set in Petersen's graph:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(sum([b[v] for v in g]))
```

```
sage: for (u,v) in g.edges(labels=None):
....:     p.add_constraint(b[u] + b[v], max=1)
sage: p.solve(objective_only=True)
4.0
```

TESTS:

Check that trac ticket #16497 is fixed:

```
sage: from sage.numerical.mip import MixedIntegerLinearProgram
sage: for type in ["binary", "integer"]:
....:     k = 3
....:     items = [1/5, 1/3, 2/3, 3/4, 5/7]
....:     maximum=1
....:     p=MixedIntegerLinearProgram()
....:     box=p.new_variable(nonnegative=True, **{type:True})
....:     for b in range(k):
....:          p.add_constraint(p.sum([items[i]*box[i,b] for i in range(len(items))]) <= maximum
....:     for i in range(len(items)):
....:         p.add_constraint(p.sum([box[i,b] for b in range(k)]) == 1)
....:     p.set_objective(None)
....:     _ = p.solve()
....:     box=p.get_values(box)
....:     print(all(v in ZZ for v in box.values()))
True
True
```

add_constraint (*linear_function*, *max=None*, *min=None*, *name=None*)

Adds a constraint to the `MixedIntegerLinearProgram`.

INPUT:

- `linear_function` – Four different types of arguments are admissible:
  - A linear function. In this case, one of the arguments `min` or `max` has to be specified.
  - A linear constraint of the form `A <= B`, `A >= B`, `A <= B <= C`, `A >= B >= C` or `A == B`.
  - A vector-valued linear function, see `linear_tensor`. In this case, one of the arguments `min` or `max` has to be specified.
  - An (in)equality of vector-valued linear functions, that is, elements of the space of linear functions tensored with a vector space. See `linear_tensor_constraints` for details.
- `max` – constant or `None` (default). An upper bound on the linear function. This must be a numerical value for scalar linear functions, or a vector for vector-valued linear functions. Not allowed if the `linear_function` argument is a symbolic (in)-equality.
- `min` – constant or `None` (default). A lower bound on the linear function. This must be a numerical value for scalar linear functions, or a vector for vector-valued linear functions. Not allowed if the `linear_function` argument is a symbolic (in)-equality.
- `name` – A name for the constraint.

To set a lower and/or upper bound on the variables use the methods `set_min` and/or `set_max` of `MixedIntegerLinearProgram`.

EXAMPLE:

Consider the following linear program:

```
Maximize:
  x + 5 * y
Constraints:
  x + 0.2 y       <= 4
  1.5 * x + 3 * y <= 4
Variables:
  x is Real (min = 0, max = None)
  y is Real (min = 0, max = None)
```

It can be solved as follows:

```
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 5*x[1])
sage: p.add_constraint(x[0] + 0.2*x[1], max=4)
sage: p.add_constraint(1.5*x[0] + 3*x[1], max=4)
sage: p.solve()     # rel tol 1e-15
6.666666666666666
```

There are two different ways to add the constraint `x[5] + 3*x[7] <= x[6] + 3` to a `MixedIntegerLinearProgram`.

The first one consists in giving `add_constraint` this very expression:

```
sage: p.add_constraint(x[5] + 3*x[7] <= x[6] + 3)
```

The second (slightly more efficient) one is to use the arguments `min` or `max`, which can only be numerical values:

```
sage: p.add_constraint(x[5] + 3*x[7] - x[6], max=3)
```

One can also define double-bounds or equality using symbols <=, >= and ==:

```
sage: p.add_constraint(x[5] + 3*x[7] == x[6] + 3)
sage: p.add_constraint(x[5] + 3*x[7] <= x[6] + 3 <= x[8] + 27)
```

Using this notation, the previous program can be written as:

```
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 5*x[1])
sage: p.add_constraint(x[0] + 0.2*x[1] <= 4)
sage: p.add_constraint(1.5*x[0] + 3*x[1] <= 4)
sage: p.solve()     # rel tol 1e-15
6.666666666666666
```

The two constraints can alse be combined into a single vector-valued constraint:

```
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 5*x[1])
sage: f_vec = vector([1, 1.5]) * x[0] + vector([0.2, 3]) * x[1];  f_vec
(1.0, 1.5)*x_0 + (0.2, 3.0)*x_1
sage: p.add_constraint(f_vec, max=vector([4, 4]))
sage: p.solve()     # rel tol 1e-15
6.666666666666666
```

Instead of specifying the maximum in the optional `max` argument, we can also use (in)equality notation for vector-valued linear functions:

```
sage: f_vec <= 4      # constant rhs becomes vector
(1.0, 1.5)*x_0 + (0.2, 3.0)*x_1 <= (4.0, 4.0)
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 5*x[1])
sage: p.add_constraint(f_vec <= 4)
sage: p.solve()       # rel tol 1e-15
6.666666666666666
```

Finally, one can use the matrix * `MIPVariable` notation to write vector-valued linear functions:

```
sage: m = matrix([[1.0, 0.2], [1.5, 3.0]]);   m
[ 1.00000000000000 0.200000000000000]
[ 1.50000000000000  3.00000000000000]
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 5*x[1])
sage: p.add_constraint(m * x <= 4)
sage: p.solve()       # rel tol 1e-15
6.666666666666666
```

TESTS:

Complex constraints:

```
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: b = p.new_variable(nonnegative=True)
sage: p.add_constraint( b[8] - b[15] <= 3*b[8] + 9)
sage: p.show()
Maximization:

Constraints:
  -2.0 x_0 - x_1 <= 9.0
Variables:
  x_0 is a continuous variable (min=0.0, max=+oo)
  x_1 is a continuous variable (min=0.0, max=+oo)
```

Empty constraint:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(sum([]),min=2)
```

Min/Max are numerical

```
sage: v = p.new_variable(nonnegative=True)
sage: p.add_constraint(v[3] + v[5], min = v[6])
Traceback (most recent call last):
...
ValueError: min and max arguments are required to be constants
sage: p.add_constraint(v[3] + v[5], max = v[6])
Traceback (most recent call last):
...
ValueError: min and max arguments are required to be constants
```

Do not add redundant elements (notice only one copy of each constraint is added):

```
sage: lp = MixedIntegerLinearProgram(solver="GLPK", check_redundant=True)
sage: for each in xrange(10): lp.add_constraint(lp[0]-lp[1],min=1)
sage: lp.show()
Maximization:
```

```
Constraints:
  1.0 <= x_0 - x_1
Variables:
  x_0 is a continuous variable (min=-oo, max=+oo)
  x_1 is a continuous variable (min=-oo, max=+oo)
```

We check for constant multiples of constraints as well:

```
sage: for each in xrange(10): lp.add_constraint(2*lp[0]-2*lp[1],min=2)
sage: lp.show()
Maximization:

Constraints:
  1.0 <= x_0 - x_1
Variables:
  x_0 is a continuous variable (min=-oo, max=+oo)
  x_1 is a continuous variable (min=-oo, max=+oo)
```

But if the constant multiple is negative, we should add it anyway (once):

```
sage: for each in xrange(10): lp.add_constraint(-2*lp[0]+2*lp[1],min=-2)
sage: lp.show()
Maximization:

Constraints:
  1.0 <= x_0 - x_1
  -2.0 <= -2.0 x_0 + 2.0 x_1
Variables:
  x_0 is a continuous variable (min=-oo, max=+oo)
  x_1 is a continuous variable (min=-oo, max=+oo)
```

Catch `True` / `False` as INPUT (trac ticket #13646):

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(True)
Traceback (most recent call last):
...
ValueError: argument must be a linear function or constraint, got True
```

base_ring()

Return the base ring.

OUTPUT:

A ring. The coefficients that the chosen solver supports.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: p.base_ring()
Real Double Field
sage: p = MixedIntegerLinearProgram(solver='ppl')
sage: p.base_ring()
Rational Field
```

best_known_objective_bound()

Return the value of the currently best known bound.

This method returns the current best upper (resp. lower) bound on the optimal value of the objective function in a maximization (resp. minimization) problem. It is equal to the output of :meth:get_objective_value if the MILP found an optimal solution, but it can differ if it was interrupted manually or after a time limit (cf :meth:solver_parameter).

Note: Has no meaning unless `solve` has been called before.

EXAMPLE:

```
sage: g = graphs.CubeGraph(9)
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: p.solver_parameter("mip_gap_tolerance",100)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(p.sum(b[v] for v in g))
sage: for v in g:
....:     p.add_constraint(b[v]+p.sum(b[u] for u in g.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1) # Force an easy non-0 solution
sage: p.solve() # rel tol 100
1.0
sage: p.best_known_objective_bound() # random
48.0
```

constraints(*indices=None*)

Returns a list of constraints, as 3-tuples.

INPUT:

- `indices` – select which constraint(s) to return
  - If `indices = None`, the method returns the list of all the constraints.
  - If `indices` is an integer i, the method returns constraint i.
  - If `indices` is a list of integers, the method returns the list of the corresponding constraints.

OUTPUT:

Each constraint is returned as a triple `lower_bound, (indices, coefficients), upper_bound`. For each of those entries, the corresponding linear function is the one associating to variable `indices[i]` the coefficient `coefficients[i]`, and 0 to all the others.

`lower_bound` and `upper_bound` are numerical values.

EXAMPLE:

First, let us define a small LP:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(p[0] - p[2], min = 1, max = 4)
sage: p.add_constraint(p[0] - 2*p[1], min = 1)
```

To obtain the list of all constraints:

```
sage: p.constraints()          # not tested
[(1.0, ([1, 0], [-1.0, 1.0]), 4.0), (1.0, ([2, 0], [-2.0, 1.0]), None)]
```

Or constraint 0 only:

```
sage: p.constraints(0)          # not tested
(1.0, ([1, 0], [-1.0, 1.0]), 4.0)
```

A list of constraints containing only 1:

```
sage: p.constraints([1])       # not tested
[(1.0, ([2, 0], [-2.0, 1.0]), None)]
```

TESTS:

As the ordering of the variables in each constraint depends on the solver used, we define a short function reordering it before it is printed. The output would look the same without this function applied:

```
sage: def reorder_constraint((lb,(ind,coef),ub)):
....:     d = dict(zip(ind, coef))
....:     ind.sort()
....:     return (lb, (ind, [d[i] for i in ind]), ub)
```

Running the examples from above, reordering applied:

```
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: p.add_constraint(p[0] - p[2], min = 1, max = 4)
sage: p.add_constraint(p[0] - 2*p[1], min = 1)
sage: sorted(map(reorder_constraint,p.constraints()))
[(1.0, ([0, 1], [1.0, -1.0]), 4.0), (1.0, ([0, 2], [1.0, -2.0]), None)]
sage: reorder_constraint(p.constraints(0))
(1.0, ([0, 1], [1.0, -1.0]), 4.0)
sage: sorted(map(reorder_constraint,p.constraints([1])))
[(1.0, ([0, 2], [1.0, -2.0]), None)]
```

gen (i)

Return the linear variable x_i.

OUTPUT:

sage: mip = MixedIntegerLinearProgram() sage: mip.gen(0) x_0 sage: [mip.gen(i) for i in range(10)] [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9]

get_backend ()

Returns the backend instance used.

This might be useful when access to additional functions provided by the backend is needed.

EXAMPLE:

This example uses the simplex algorthm and prints information:

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x, y = p[0], p[1]
sage: p.add_constraint(2*x + 3*y, max = 6)
sage: p.add_constraint(3*x + 2*y, max = 6)
sage: p.set_objective(x + y + 7)
sage: b = p.get_backend()
sage: b.solver_parameter("simplex_or_intopt", "simplex_only")
sage: b.solver_parameter("verbosity_simplex", "GLP_MSG_ALL")
sage: p.solve()  # rel tol 1e-5
GLPK Simplex Optimizer, v4.55
2 rows, 2 columns, 4 non-zeros
*     0: obj =   7.000000000e+00  infeas =  0.000e+00 (0)
*     2: obj =   9.400000000e+00  infeas =  0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
9.4
```

get_max (v)

Returns the maximum value of a variable.

INPUT:

- v – a variable.

OUTPUT:

Maximum value of the variable, or None if the variable has no upper bound.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_max(v[1])
sage: p.set_max(v[1],6)
sage: p.get_max(v[1])
6.0
```

get_min(*v*)

Returns the minimum value of a variable.

INPUT:

- v – a variable

OUTPUT:

Minimum value of the variable, or None if the variable has no lower bound.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_min(v[1])
0.0
sage: p.set_min(v[1],6)
sage: p.get_min(v[1])
6.0
sage: p.set_min(v[1], None)
sage: p.get_min(v[1])
```

get_objective_value()

Return the value of the objective function.

Note: Behaviour is undefined unless solve has been called before.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x, y = p[0], p[1]
sage: p.add_constraint(2*x + 3*y, max = 6)
sage: p.add_constraint(3*x + 2*y, max = 6)
sage: p.set_objective(x + y + 7)
sage: p.solve()  # rel tol 1e-5
9.4
sage: p.get_objective_value()  # rel tol 1e-5
9.4
```

get_relative_objective_gap()

Return the relative objective gap of the best known solution.

For a minimization problem, this value is computed by $(\texttt{bestinteger} - \texttt{bestobjective})/(1e{-}10 + |\texttt{bestobjective}|)$, where bestinteger is the value returned by get_objective_value()

and `bestobjective` is the value returned by `best_known_objective_bound()`. For a maximization problem, the value is computed by $(\texttt{bestobjective} - \texttt{bestinteger})/(1e-10+|\texttt{bestobjective}|)$.

Note: Has no meaning unless `solve` has been called before.

EXAMPLE:

```
sage: g = graphs.CubeGraph(9)
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: p.solver_parameter("mip_gap_tolerance",100)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(p.sum(b[v] for v in g))
sage: for v in g:
....:     p.add_constraint(b[v]+p.sum(b[u] for u in g.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1) # Force an easy non-0 solution
sage: p.solve() # rel tol 100
1.0
sage: p.get_relative_objective_gap() # random
46.99999999999999
```

TESTS:

Just make sure that the variable *has* been defined, and is not just undefined:

```
sage: p.get_relative_objective_gap() > 1
True
```

get_values(**lists*)

Return values found by the previous call to `solve()`.

INPUT:

- Any instance of `MIPVariable` (or one of its elements), or lists of them.

OUTPUT:

- Each instance of `MIPVariable` is replaced by a dictionary containing the numerical values found for each corresponding variable in the instance.
- Each element of an instance of a `MIPVariable` is replaced by its corresponding numerical value.

Note: While a variable may be declared as binary or integer, its value as returned by the solver is of type `float`.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
sage: y = p.new_variable(nonnegative=True)
sage: p.set_objective(x[3] + 3*y[2,9] + x[5])
sage: p.add_constraint(x[3] + y[2,9] + 2*x[5], max=2)
sage: p.solve()
6.0
```

To return the optimal value of `y[2,9]`:

```
sage: p.get_values(y[2,9])
2.0
```

To get a dictionary identical to `x` containing optimal values for the corresponding variables

```
sage: x_sol = p.get_values(x)
sage: x_sol.keys()
[3, 5]
```

Obviously, it also works with variables of higher dimension:

```
sage: y_sol = p.get_values(y)
```

We could also have tried

```
sage: [x_sol, y_sol] = p.get_values(x, y)
```

Or:

```
sage: [x_sol, y_sol] = p.get_values([x, y])
```

is_binary(*e*)

Tests whether the variable `e` is binary. Variables are real by default.

INPUT:

- `e` – A variable (not a `MIPVariable`, but one of its elements.)

OUTPUT:

`True` if the variable `e` is binary; `False` otherwise.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.is_binary(v[1])
False
sage: p.set_binary(v[1])
sage: p.is_binary(v[1])
True
```

is_integer(*e*)

Tests whether the variable is an integer. Variables are real by default.

INPUT:

- `e` – A variable (not a `MIPVariable`, but one of its elements.)

OUTPUT:

`True` if the variable `e` is an integer; `False` otherwise.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.is_integer(v[1])
False
sage: p.set_integer(v[1])
sage: p.is_integer(v[1])
True
```

is_real(*e*)

Tests whether the variable is real.

INPUT:

- e – A variable (not a `MIPVariable`, but one of its elements.)

OUTPUT:

`True` if the variable is real; `False` otherwise.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.is_real(v[1])
True
sage: p.set_binary(v[1])
sage: p.is_real(v[1])
False
sage: p.set_real(v[1])
sage: p.is_real(v[1])
True
```

linear_constraints_parent()

Return the parent for all linear constraints

See `linear_functions` for more details.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: p.linear_constraints_parent()
Linear constraints over Real Double Field
```

linear_function(*x*)

Construct a new linear function

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: p.linear_function({1:3, 4:5})
3*x_1 + 5*x_4
```

This is equivalent to:

```
sage: p({1:3, 4:5})
3*x_1 + 5*x_4
```

linear_functions_parent()

Return the parent for all linear functions

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: p.linear_functions_parent()
Linear functions over Real Double Field
```

new_variable(*real=False*, *binary=False*, *integer=False*, *nonnegative=False*, *name=''*)

Return a new MIPVariable

A new variable `x` is defined by:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
```

It behaves exactly as a usual dictionary would. It can use any key argument you may like, as `x[5]` or `x["b"]`, and has methods `items()` and `keys()`.

See also:

- `set_min()`, `get_min()` – set/get the lower bound of a variable. Note that by default, all variables are non-negative.
- `set_max()`, `get_max()` – set/get the upper bound of a variable.

INPUT:

- `binary, integer, real` – boolean. Set one of these arguments to `True` to ensure that the variable gets the corresponding type.
- `nonnegative` – boolean, default `False`. Whether the variable should be assumed to be nonnegative. Rather useless for the binary type.
- `name` – string. Associates a name to the variable. This is only useful when exporting the linear program to a file using `write_mps` or `write_lp`, and has no other effect.

OUTPUT:

A new instance of `MIPVariable` associated to the current `MixedIntegerLinearProgram`.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(); x
MIPVariable of dimension 1
sage: x0 = x[0]; x0
x_0
```

By default, variables are unbounded:

```
    sage: print p.get_min(x0)
    None
    sage: print p.get_max(x0)
    None

To define two dictionaries of variables, the first being
of real type, and the second of integer type ::

    sage: x = p.new_variable(real=True, nonnegative=True)
    sage: y = p.new_variable(integer=True, nonnegative=True)
    sage: p.add_constraint(x[2] + y[3,5], max=2)
    sage: p.is_integer(x[2])
    False
    sage: p.is_integer(y[3,5])
    True
```

An exception is raised when two types are supplied

```
sage: z = p.new_variable(real = True, integer = True)
Traceback (most recent call last):
...
ValueError: Exactly one of the available types has to be True
```

Unbounded variables:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(real=True)
sage: y = p.new_variable(integer=True)
sage: p.add_constraint(x[0]+x[3] <= 8)
sage: p.add_constraint(y[0] >= y[1])
```

```
sage: p.show()
Maximization:

Constraints:
  x_0 + x_1 <= 8.0
  - x_2 + x_3 <= 0.0
Variables:
  x_0 is a continuous variable (min=-oo, max=+oo)
  x_1 is a continuous variable (min=-oo, max=+oo)
  x_2 is an integer variable (min=-oo, max=+oo)
  x_3 is an integer variable (min=-oo, max=+oo)
```

On the Sage command line, generator syntax is accepted as a shorthand for generating new variables with default settings:

```
sage: mip.<x, y, z> = MixedIntegerLinearProgram()
sage: mip.add_constraint(x[0] + y[1] + z[2] <= 10)
sage: mip.show()
Maximization:

Constraints:
  x[0] + y[1] + z[2] <= 10.0
Variables:
  x[0] = x_0 is a continuous variable (min=-oo, max=+oo)
  y[1] = x_1 is a continuous variable (min=-oo, max=+oo)
  z[2] = x_2 is a continuous variable (min=-oo, max=+oo)
```

number_of_constraints()

Returns the number of constraints assigned so far.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(p[0] - p[2], min = 1, max = 4)
sage: p.add_constraint(p[0] - 2*p[1], min = 1)
sage: p.number_of_constraints()
2
```

number_of_variables()

Returns the number of variables used so far.

Note that this is backend-dependent, i.e. we count solver's variables rather than user's variables. An example of the latter can be seen below: Gurobi converts double inequalities, i.e. inequalities like $m <= c^T x <= M$, with $m < M$, into equations, by adding extra variables: $c^T x + y = M, 0 <= y <= M - m$.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(p[0] - p[2], max = 4)
sage: p.number_of_variables()
2
sage: p.add_constraint(p[0] - 2*p[1], min = 1)
sage: p.number_of_variables()
3
sage: p = MixedIntegerLinearProgram(solver="glpk")
sage: p.add_constraint(p[0] - p[2], min = 1, max = 4)
sage: p.number_of_variables()
2
sage: p = MixedIntegerLinearProgram(solver="gurobi")    # optional - Gurobi
sage: p.add_constraint(p[0] - p[2], min = 1, max = 4)  # optional - Gurobi
```

```
sage: p.number_of_variables()                        # optional - Gurobi
3
```

polyhedron(***kwds*)

Returns the polyhedron defined by the Linear Program.

INPUT:

All arguments given to this method are forwarded to the constructor of the `Polyhedron()` class.

OUTPUT:

A `Polyhedron()` object whose i-th variable represents the i-th variable of `self`.

> **Warning:** The polyhedron is built from the variables stored by the LP solver (i.e. the output of `show()`). While they usually match the ones created explicitly when defining the LP, a solver like Gurobi has been known to introduce additional variables to store constraints of the type `lower_bound <= linear_function <= upper bound`. You should be fine if you did not install Gurobi or if you do not use it as a solver, but keep an eye on the number of variables in the polyhedron, or on the output of `show()`. Just in case.

See also:

`to_linear_program()` – return the `MixedIntegerLinearProgram` object associated with a `Polyhedron()` object.

EXAMPLES:

A LP on two variables:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(0 <= 2*p['x'] + p['y'] <= 1)
sage: p.add_constraint(0 <= 3*p['y'] + p['x'] <= 2)
sage: P = p.polyhedron(); P
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

3-D Polyhedron:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(0 <= 2*p['x'] + p['y'] + 3*p['z'] <= 1)
sage: p.add_constraint(0 <= 2*p['y'] + p['z'] + 3*p['x'] <= 1)
sage: p.add_constraint(0 <= 2*p['z'] + p['x'] + 3*p['y'] <= 1)
sage: P = p.polyhedron(); P
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
```

An empty polyhedron:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.add_constraint(2*v['x'] + v['y'] + 3*v['z'] <= 1)
sage: p.add_constraint(2*v['y'] + v['z'] + 3*v['x'] <= 1)
sage: p.add_constraint(2*v['z'] + v['x'] + 3*v['y'] >= 2)
sage: P = p.polyhedron(); P
The empty polyhedron in QQ^3
```

An unbounded polyhedron:

```
sage: p = MixedIntegerLinearProgram()
sage: p.add_constraint(2*p['x'] + p['y'] - p['z'] <= 1)
sage: P = p.polyhedron(); P
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 1 ray, 2 lines
```

A square (see trac ticket #14395)

```
sage: p = MixedIntegerLinearProgram()
sage: x,y = p['x'], p['y']
sage: p.add_constraint( x <= 1 )
sage: p.add_constraint( x >= -1 )
sage: p.add_constraint( y <= 1 )
sage: p.add_constraint( y >= -1 )
sage: p.polyhedron()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

remove_constraint (*i*)

Removes a constraint from self.

INPUT:

- `i` – Index of the constraint to remove.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x, y = p[0], p[1]
sage: p.add_constraint(x + y, max = 10)
sage: p.add_constraint(x - y, max = 0)
sage: p.add_constraint(x, max = 4)
sage: p.show()
Maximization:

Constraints:
  x_0 + x_1 <= 10.0
  x_0 - x_1 <= 0.0
  x_0 <= 4.0
...
sage: p.remove_constraint(1)
sage: p.show()
Maximization:

Constraints:
  x_0 + x_1 <= 10.0
  x_0 <= 4.0
...
sage: p.number_of_constraints()
2
```

remove_constraints (*constraints*)

Remove several constraints.

INPUT:

- `constraints` – an iterable containing the indices of the rows to remove.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x, y = p[0], p[1]
sage: p.add_constraint(x + y, max = 10)
sage: p.add_constraint(x - y, max = 0)
sage: p.add_constraint(x, max = 4)
sage: p.show()
Maximization:
```

```
Constraints:
  x_0 + x_1 <= 10.0
  x_0 - x_1 <= 0.0
  x_0 <= 4.0
...
sage: p.remove_constraints([0, 1])
sage: p.show()
Maximization:

Constraints:
  x_0 <= 4.0
...
sage: p.number_of_constraints()
1
```

When checking for redundant constraints, make sure you remove only the constraints that were actually added. Problems could arise if you have a function that builds lps non-interactively, but it fails to check whether adding a constraint actually increases the number of constraints. The function might later try to remove constraints that are not actually there:

```
sage: p = MixedIntegerLinearProgram(check_redundant=True)
sage: x, y = p[0], p[1]
sage: p.add_constraint(x + y, max = 10)
sage: for each in xrange(10): p.add_constraint(x - y, max = 10)
sage: p.add_constraint(x, max = 4)
sage: p.number_of_constraints()
3
sage: p.remove_constraints(range(1,9))
Traceback (most recent call last):
...
IndexError: pop index out of range
sage: p.remove_constraint(1)
sage: p.number_of_constraints()
2
```

We should now be able to add the old constraint back in:

```
sage: for each in xrange(10): p.add_constraint(x - y, max = 10)
sage: p.number_of_constraints()
3
```

set_binary(*ee*)

Sets a variable or a `MIPVariable` as binary.

INPUT:

- `ee` – An instance of `MIPVariable` or one of its elements.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
```

With the following instruction, all the variables from x will be binary:

```
sage: p.set_binary(x)
sage: p.set_objective(x[0] + x[1])
sage: p.add_constraint(-3*x[0] + 2*x[1], max=2)
```

It is still possible, though, to set one of these variables as integer while keeping the others as they are:

```
sage: p.set_integer(x[3])
```

set_integer(*ee*)

Sets a variable or a `MIPVariable` as integer.

INPUT:

- `ee` – An instance of `MIPVariable` or one of its elements.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
```

With the following instruction, all the variables from x will be integers:

```
sage: p.set_integer(x)
sage: p.set_objective(x[0] + x[1])
sage: p.add_constraint(-3*x[0] + 2*x[1], max=2)
```

It is still possible, though, to set one of these variables as binary while keeping the others as they are:

```
sage: p.set_binary(x[3])
```

set_max(*v*, *max*)

Sets the maximum value of a variable.

INPUT

- `v` – a variable.
- `max` – the maximum value the variable can take. When `max=None`, the variable has no upper bound.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_max(v[1])
sage: p.set_max(v[1],6)
sage: p.get_max(v[1])
6.0
```

With a `MIPVariable` as an argument:

```
sage: vv = p.new_variable(real=True)
sage: p.get_max(vv)
sage: p.get_max(vv[0])
sage: p.set_max(vv,5)
sage: p.get_max(vv[0])
5.0
sage: p.get_max(vv[9])
5.0
```

set_min(*v*, *min*)

Sets the minimum value of a variable.

INPUT:

- `v` – a variable.
- `min` – the minimum value the variable can take. When `min=None`, the variable has no lower bound.

See also:

- `get_min()` – get the minimum value of a variable.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_min(v[1])
0.0
sage: p.set_min(v[1],6)
sage: p.get_min(v[1])
6.0
sage: p.set_min(v[1], None)
sage: p.get_min(v[1])
```

With a `MIPVariable` as an argument:

```
sage: vv = p.new_variable(real=True)
sage: p.get_min(vv)
sage: p.get_min(vv[0])
sage: p.set_min(vv,5)
sage: p.get_min(vv[0])
5.0
sage: p.get_min(vv[9])
5.0
```

set_objective(*obj*)

Sets the objective of the `MixedIntegerLinearProgram`.

INPUT:

- `obj` – A linear function to be optimized. (can also be set to `None` or `0` when just looking for a feasible solution)

EXAMPLE:

Let's solve the following linear program:

```
Maximize:
  x + 5 * y
Constraints:
  x + 0.2 y       <= 4
  1.5 * x + 3 * y <= 4
Variables:
  x is Real (min = 0, max = None)
  y is Real (min = 0, max = None)
```

This linear program can be solved as follows:

```
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[1] + 5*x[2])
sage: p.add_constraint(x[1] + 2/10*x[2], max=4)
sage: p.add_constraint(1.5*x[1]+3*x[2], max=4)
sage: round(p.solve(),5)
6.66667
sage: p.set_objective(None)
sage: _ = p.solve()
```

set_problem_name(*name*)

Sets the name of the `MixedIntegerLinearProgram`.

INPUT:

- `name` – A string representing the name of the `MixedIntegerLinearProgram`.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: p.set_problem_name("Test program")
sage: p
Mixed Integer Program "Test program" ( maximization, 0 variables, 0 constraints )
```

set_real(*ee*)

Sets a variable or a `MIPVariable` as real.

INPUT:

- `ee` – An instance of `MIPVariable` or one of its elements.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=True)
```

With the following instruction, all the variables from x will be real:

```
sage: p.set_real(x)
sage: p.set_objective(x[0] + x[1])
sage: p.add_constraint(-3*x[0] + 2*x[1], max=2)

It is still possible, though, to set one of these
variables as binary while keeping the others as they are::

   sage: p.set_binary(x[3])
```

show()

Displays the `MixedIntegerLinearProgram` in a human-readable way.

EXAMPLES:

When constraints and variables have names

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x = p.new_variable(name="Hey")
sage: p.set_objective(x[1] + x[2])
sage: p.add_constraint(-3*x[1] + 2*x[2], max=2, name="Constraint_1")
sage: p.show()
Maximization:
  Hey[1] + Hey[2]
Constraints:
  Constraint_1: -3.0 Hey[1] + 2.0 Hey[2] <= 2.0
Variables:
  Hey[1] = x_0 is a continuous variable (min=-oo, max=+oo)
  Hey[2] = x_1 is a continuous variable (min=-oo, max=+oo)
```

Without any names

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[1] + x[2])
sage: p.add_constraint(-3*x[1] + 2*x[2], max=2)
```

```
sage: p.show()
Maximization:
  x_0 + x_1
Constraints:
  -3.0 x_0 + 2.0 x_1 <= 2.0
Variables:
  x_0 is a continuous variable (min=0.0, max=+oo)
  x_1 is a continuous variable (min=0.0, max=+oo)
```

With **Q** coefficients:

```
sage: p = MixedIntegerLinearProgram(solver='ppl')
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[1] + 1/2*x[2])
sage: p.add_constraint(-3/5*x[1] + 2/7*x[2], max=2/5)
sage: p.show()
Maximization:
  x_0 + 1/2 x_1
Constraints:
  constraint_0: -3/5 x_0 + 2/7 x_1 <= 2/5
Variables:
  x_0 is a continuous variable (min=0, max=+oo)
  x_1 is a continuous variable (min=0, max=+oo)
```

solve (*log=None*, *objective_only=False*)

Solves the `MixedIntegerLinearProgram`.

INPUT:

- `log` – integer (default: `None`) The verbosity level. Indicates whether progress should be printed during computation. The solver is initialized to report no progress.
- `objective_only` – Boolean variable.
  - When set to `True`, only the objective function is returned.
  - When set to `False` (default), the optimal numerical values are stored (takes computational time).

OUTPUT:

The optimal value taken by the objective function.

> **Warning:** By default, all variables of a LP are assumed to be non-negative. See `set_min()` to change it.

EXAMPLES:

Consider the following linear program:

```
Maximize:
  x + 5 * y
Constraints:
  x + 0.2 y       <= 4
  1.5 * x + 3 * y <= 4
Variables:
  x is Real (min = 0, max = None)
  y is Real (min = 0, max = None)
```

This linear program can be solved as follows:

```
    sage: p = MixedIntegerLinearProgram(maximization=True)
    sage: x = p.new_variable(nonnegative=True)
    sage: p.set_objective(x[1] + 5*x[2])
    sage: p.add_constraint(x[1] + 0.2*x[2], max=4)
    sage: p.add_constraint(1.5*x[1] + 3*x[2], max=4)
    sage: round(p.solve(),6)
    6.666667
    sage: x = p.get_values(x)
    sage: round(x[1],6) # abs tol 1e-15
    0.0
    sage: round(x[2],6)
    1.333333

Computation of a maximum stable set in Petersen's graph::

    sage: g = graphs.PetersenGraph()
    sage: p = MixedIntegerLinearProgram(maximization=True)
    sage: b = p.new_variable(nonnegative=True)
    sage: p.set_objective(sum([b[v] for v in g]))
    sage: for (u,v) in g.edges(labels=None):
    ...       p.add_constraint(b[u] + b[v], max=1)
    sage: p.set_binary(b)
    sage: p.solve(objective_only=True)
    4.0
```

Constraints in the objective function are respected:

```
sage: p = MixedIntegerLinearProgram()
sage: x, y = p[0], p[1]
sage: p.add_constraint(2*x + 3*y, max = 6)
sage: p.add_constraint(3*x + 2*y, max = 6)
sage: p.set_objective(x + y + 7)
sage: p.set_integer(x); p.set_integer(y)
sage: p.solve()
9.0
```

solver_parameter (*name*, *value=None*)

Return or define a solver parameter

The solver parameters are by essence solver-specific, which means their meaning heavily depends on the solver used.

(If you do not know which solver you are using, then you use GLPK).

Aliases:

Very common parameters have aliases making them solver-independent. For example, the following:

```
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: p.solver_parameter("timelimit", 60)
```

Sets the solver to stop its computations after 60 seconds, and works with GLPK, CPLEX and Gurobi.

- `"timelimit"` – defines the maximum time spent on a computation. Measured in seconds.

Another example is the `"logfile"` parameter, which is used to specify the file in which computation logs are recorded. By default, the logs are not recorded, and we can disable this feature providing an empty filename. This is currently working with CPLEX and Gurobi:

```
sage: p = MixedIntegerLinearProgram(solver = "CPLEX") # optional - CPLEX
sage: p.solver_parameter("logfile")                   # optional - CPLEX
```

```
''
sage: p.solver_parameter("logfile", "/dev/null")       # optional - CPLEX
sage: p.solver_parameter("logfile")                    # optional - CPLEX
'/dev/null'
sage: p.solver_parameter("logfile", '')                # optional - CPLEX
sage: p.solver_parameter("logfile")                    # optional - CPLEX
''
```

Solver-specific parameters:

- GLPK : We have implemented very close to comprehensive coverage of the GLPK solver parameters for the simplex and integer optimization methods. For details, see the documentation of `GLPKBackend.solver_parameter`.
- CPLEX's parameters are identified by a string. Their list is available on ILOG's website.

  The command

  ```
  sage: p = MixedIntegerLinearProgram(solver = "CPLEX") # optional - CPLEX
  sage: p.solver_parameter("CPX_PARAM_TILIM", 60)       # optional - CPLEX
  ```

  works as intended.
- Gurobi's parameters should all be available through this method. Their list is available on Gurobi's website http://www.gurobi.com/documentation/5.5/reference-manual/node798.

INPUT:

- `name` (string) – the parameter
- `value` – the parameter's value if it is to be defined, or `None` (default) to obtain its current value.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: p.solver_parameter("timelimit", 60)
sage: p.solver_parameter("timelimit")
60.0
```

sum (*L*)

Efficiently computes the sum of a sequence of `LinearFunction` elements

INPUT:

- `mip` – the `MixedIntegerLinearProgram` parent.
- `L` – list of `LinearFunction` instances.

Note: The use of the regular `sum` function is not recommended as it is much less efficient than this one

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(nonnegative=True)
```

The following command:

```
sage: s = p.sum([v[i] for i in xrange(90)])
```

is much more efficient than:

```
sage: s = sum([v[i] for i in xrange(90)])
```

write_lp (*filename*)

Write the linear program as a LP file.

This function export the problem as a LP file.

INPUT:

- `filename` – The file in which you want the problem to be written.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[1] + x[2])
sage: p.add_constraint(-3*x[1] + 2*x[2], max=2)
sage: p.write_lp(os.path.join(SAGE_TMP, "lp_problem.lp"))
Writing problem data to ...
9 lines were written
```

For more information about the LP file format : http://lpsolve.sourceforge.net/5.5/lp-format.htm

write_mps (*filename*, *modern=True*)

Write the linear program as a MPS file.

This function export the problem as a MPS file.

INPUT:

- `filename` – The file in which you want the problem to be written.
- `modern` – Lets you choose between Fixed MPS and Free MPS
  - `True` – Outputs the problem in Free MPS
  - `False` – Outputs the problem in Fixed MPS

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[1] + x[2])
sage: p.add_constraint(-3*x[1] + 2*x[2], max=2,name="OneConstraint")
sage: p.write_mps(os.path.join(SAGE_TMP, "lp_problem.mps"))
Writing problem data to ...
17 records were written
```

For information about the MPS file format : http://en.wikipedia.org/wiki/MPS_%28format%29

CHAPTER

THREE

SEMIDEFINITE PROGRAMMING

A semidefinite program (SDP) is an optimization problem in the following form

$$\max \sum_{i,j=1}^{n} c_{ij}x_{ij}$$
$$\text{Subject to:} \sum_{i,j=1}^{n} a_{ijk}x_{ij} = b_k, \qquad k = 1 \dots m$$
$$X \succeq 0$$

where the x_{ij}, $i \le i, j \le n$ are n^2 variables satisfying the symmetry conditions $x_{ij} = x_{ji}$ for all i, j, the c_{ij}, a_{ijk} and b_k are real coefficients, and X is positive semidefinite, i.e., all the eigenvalues of X are nonnegative.

A wide variety of problems in optimization can be formulated in this standard form. Then, solvers are able to calculate a solution.

For instance, you want to minimize $x_0 - x_1$ where:

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} x_0 + \begin{pmatrix} 3 & 4 \\ 4 & 5 \end{pmatrix} x_1 \preceq \begin{pmatrix} 5 & 6 \\ 6 & 7 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} x_0 + \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} x_1 \preceq \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}, \quad x_0 \ge 0, x_1 \ge 0.$$

A semidefinite program can give you an answer to the problem above. Here is how it's done:

1. You have to create an instance of `SemidefiniteProgram`. We add the parameter maximization=False since we want to minimize $x_0 - x_1$.
2. Create an dictionary `x` of integer variables `x` via `x = p.new_variable()` (note that **by default all variables are non-negative**, cf `new_variable()`).
3. Add those two inequalities as inequality constraints via `add_constraint`.
4. Specify the objective function via `set_objective`. In our case it is $x_0 - x_1$. If it is a pure constraint satisfaction problem, specify it as `None`.
5. To check if everything is set up correctly, you can print the problem via `show`.
6. `Solve` it and print the solution.

The following example shows all these steps:

```
sage: p = SemidefiniteProgram(maximization = False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1])
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 2.]])
```

```
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a3)
sage: p.add_constraint(b1*x[0] + b2*x[1] <= b3)
sage: p.solver_parameter("show_progress", True)
sage: print 'Objective Value:', round(p.solve(),3)
Objective Value:      pcost       dcost       gap    pres   dres   k/t
0: -3.00...
...
Optimal solution found.
-3.0
sage: map(lambda x: round(x,3), p.get_values(x).itervalues())
[-1.0, 2.0]
sage: p.show()
Minimization:
  x_0 - x_1
Constraints:
  constraint_0: [1.0 2.0][2.0 3.0]x_0 + [3.0 4.0][4.0 5.0]x_1 <=  [5.0 6.0][6.0 7.0]
  constraint_1: [1.0 1.0][1.0 1.0]x_0 + [2.0 2.0][2.0 2.0]x_1 <=  [3.0 3.0][3.0 3.0]
Variables:
   x_0,  x_1
```

More interesting example, the `Lovasz theta` of the 7-gon:

```
sage: c=graphs.CycleGraph(7)
sage: c2=c.distance_graph(2).adjacency_matrix()
sage: c3=c.distance_graph(3).adjacency_matrix()
sage: p.<y>=SemidefiniteProgram()
sage: p.add_constraint((1/7)*matrix.identity(7)>=-y[0]*c2-y[1]*c3)
sage: p.set_objective(y[0]*(c2**2).trace()+y[1]*(c3**2).trace())
sage: x=p.solve(); x+1
3.31766...
```

The default CVXOPT backend computes with the Real Double Field, for example:

```
sage: p = SemidefiniteProgram(solver='cvxopt')
sage: p.base_ring()
Real Double Field
sage: x = p.new_variable()
sage: 0.5 + 3/2*x[1]
0.5 + 1.5*x_0
```

3.1 Linear Variables and Expressions

To make your code more readable, you can construct `SDPVariable` objects that can be arbitrarily named and indexed. Internally, this is then translated back to the x_i variables. For example:

```
sage: sdp.<a,b> = SemidefiniteProgram()
sage: a
SDPVariable
sage: 5 + a[1] + 2*b[3]
5 + x_0 + 2*x_1
```

Indices can be any object, not necessarily integers. Multi-indices are also allowed:

```
sage: a[4, 'string', QQ]
x_2
sage: a[4, 'string', QQ] - 7*b[2]
x_2 - 7*x_3
sage: sdp.show()
Maximization:

Constraints:
Variables:
  a[1],  b[3],  a[(4, 'string', Rational Field)],  b[2]
```

3.2 Index of functions and methods

Below are listed the methods of `SemidefiniteProgram`. This module also implements the `SDPSolverException` exception, as well as the `SDPVariable` class.

`add_constraint()`	Adds a constraint to the `SemidefiniteProgram`
`base_ring()`	Return the base ring
`get_backend()`	Returns the backend instance used
`get_values()`	Return values found by the previous call to `solve()`
`linear_constraints_parent()`	Return the parent for all linear constraints
`linear_function()`	Construct a new linear function
`linear_functions_parent()`	Return the parent for all linear functions
`new_variable()`	Returns an instance of `SDPVariable` associated
`number_of_constraints()`	Returns the number of constraints assigned so far
`number_of_variables()`	Returns the number of variables used so far
`set_objective()`	Sets the objective of the `SemidefiniteProgram`
`set_problem_name()`	Sets the name of the `SemidefiniteProgram`
`show()`	Displays the `SemidefiniteProgram` in a human-readable
`solve()`	Solves the `SemidefiniteProgram`
`solver_parameter()`	Return or define a solver parameter
`sum()`	Efficiently computes the sum of a sequence of LinearFunction elements

AUTHORS:

- Ingolfur Edvardsson (2014/08): added extension for exact computation
- Dima Pasechnik (2014, 2015) : supervision, minor fixes

exception `sage.numerical.sdp.`**`SDPSolverException`**

Bases: `exceptions.RuntimeError`

Exception raised when the solver fails.

`SDPSolverException` is the exception raised when the solver fails.

EXAMPLE:

```
sage: from sage.numerical.sdp import SDPSolverException
sage: SDPSolverException("Error")
SDPSolverException('Error',)
```

TESTS:

No solution:

```
sage: p=SemidefiniteProgram(solver="cvxopt")
sage: x=p.new_variable()
sage: p.set_objective(x[0])
sage: a = matrix([[1,2],[2,4]])
sage: b = matrix([[1,9],[9,4]])
sage: p.add_constraint( a*x[0] == b   )
sage: p.solve()
...
Traceback (most recent call last):
...
SDPSolverException: ...
```

The value of the exception:

```
sage: from sage.numerical.sdp import SDPSolverException
sage: e = SDPSolverException("Error")
sage: print e
Error
```

class `sage.numerical.sdp.`**`SDPVariable`**

Bases: `sage.structure.element.Element`

`SDPVariable` is a variable used by the class `SemidefiniteProgram`.

> **Warning:** You should not instantiate this class directly. Instead, use `SemidefiniteProgram.new_variable()`.

`items`()

Returns the pairs (keys,value) contained in the dictionary.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
sage: p.set_objective(v[0] + v[1])
sage: v.items()
[(0, x_0), (1, x_1)]
```

`keys`()

Returns the keys already defined in the dictionary.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
sage: p.set_objective(v[0] + v[1])
sage: v.keys()
[0, 1]
```

`values`()

Returns the symbolic variables associated to the current dictionary.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
sage: p.set_objective(v[0] + v[1])
sage: v.values()
[x_0, x_1]
```

class `sage.numerical.sdp.`**`SDPVariableParent`**

Bases: `sage.structure.parent.Parent`

Parent for `SDPVariable`.

> **Warning:** This class is for internal use. You should not instantiate it yourself. Use `SemidefiniteProgram.new_variable()` to generate sdp variables.

`Element`

alias of `SDPVariable`

class `sage.numerical.sdp.`**`SemidefiniteProgram`**

Bases: `sage.structure.sage_object.SageObject`

The `SemidefiniteProgram` class is the link between Sage, semidefinite programming (SDP) and semidefinite programming solvers.

A Semidefinite Programming (SDP) consists of variables, linear constraints on these variables, and an objective function which is to be maximised or minimised under these constraints.

See the Wikipedia article Semidefinite_programming for further information on semidefinite programming, and the `SDP module` for its use in Sage.

INPUT:

- `solver` – selects a solver:
  - **CVXOPT (`solver="CVXOPT"`). See the CVXOPT** web site.
  - If `solver=None` (default), the default solver is used (see `default_sdp_solver()`)
- `maximization`
  - When set to `True` (default), the `SemidefiniteProgram` is defined as a maximization.
  - When set to `False`, the `SemidefiniteProgram` is defined as a minimization.

See also:

- `default_sdp_solver()` – Returns/Sets the default SDP solver.

EXAMPLES:

Computation of a basic Semidefinite Program:

```
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1])
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 2.]])
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a3)
sage: p.add_constraint(b1*x[0] + b2*x[1] <= b3)
sage: round(p.solve(), 2)
-3.0
```

`add_constraint`(*linear_function*, *name=None*)

Adds a constraint to the `SemidefiniteProgram`.

INPUT:

- **`linear_function` – Two different types of arguments are possible:**
  - A linear function. In this case, arguments `min` or `max` have to be specified.
  - A linear constraint of the form `A <= B`, `A >= B`, `A <= B <= C`, `A >= B >= C` or `A == B`. In this case, arguments `min` and `max` will be ignored.
- `name` – A name for the constraint.

EXAMPLE:

Let's solve the following semidefinite program:

```
Maximize:
  x + 5 * y
Constraints:
  [1,2][2,3]x + [1,1][1,1] y        <= [1,-1][-1,1]
Variables:
  x, y
```

This SDP can be solved as follows:

```
sage: p = SemidefiniteProgram(maximization=True)
sage: x = p.new_variable()
sage: p.set_objective(x[1] + 5*x[2])
sage: a1 = matrix([[1,2],[2,3]])
sage: a2 = matrix([[1,1],[1,1]])
sage: a3 = matrix([[1,-1],[-1,1]])
sage: p.add_constraint(a1*x[1]+a2*x[2] <= a3)
sage: round(p.solve(),5)
16.2
```

One can also define double-bounds or equality using the symbol >= or ==:

```
sage: p = SemidefiniteProgram(maximization=True)
sage: x = p.new_variable()
sage: p.set_objective(x[1] + 5*x[2])
sage: a1 = matrix([[1,2],[2,3]])
sage: a2 = matrix([[1,1],[1,1]])
sage: a3 = matrix([[1,-1],[-1,1]])
sage: p.add_constraint(a3 >= a1*x[1] + a2*x[2])
sage: round(p.solve(),5)
16.2
```

TESTS:

Complex constraints:

```
sage: p = SemidefiniteProgram(solver = "cvxopt")
sage: b = p.new_variable()
sage: a1 = matrix([[1,2],[2,3]])
sage: a2 = matrix([[1,-2],[-2,4]])
sage: p.add_constraint(a1*b[8] - a1*b[15] <= a2*b[8])
sage: p.show()
Maximization:

Constraints:
    constraint_0: [ 0.0  4.0][ 4.0 -1.0]x_0 + [-1.0 -2.0][-2.0 -3.0]x_1 <=  [0 0][0 0]
Variables:
  x_0, x_1
```

Empty constraint:

```
sage: p=SemidefiniteProgram()
sage: p.add_constraint(sum([]))
```

base_ring()

Return the base ring.

OUTPUT:

A ring. The coefficients that the chosen solver supports.

EXAMPLES:

```
sage: p = SemidefiniteProgram(solver='cvxopt')
sage: p.base_ring()
Real Double Field
```

gen(*i*)

Return the linear variable x_i.

OUTPUT:

sage: sdp = SemidefiniteProgram() sage: sdp.gen(0) x_0 sage: [sdp.gen(i) for i in range(10)] [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9]

get_backend()

Returns the backend instance used.

This might be useful when acces to additional functions provided by the backend is needed.

EXAMPLE:

This example prints a matrix coefficient:

```
sage: p = SemidefiniteProgram(solver="cvxopt")
sage: x = p.new_variable()
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a1)
sage: b = p.get_backend()
sage: b.get_matrix()[0][0]
(
    [-1.0 -2.0]
-1, [-2.0 -3.0]
)
```

get_values(**lists*)

Return values found by the previous call to `solve()`.

INPUT:

- Any instance of `SDPVariable` (or one of its elements), or lists of them.

OUTPUT:

- Each instance of `SDPVariable` is replaced by a dictionary containing the numerical values found for each corresponding variable in the instance.
- Each element of an instance of a `SDPVariable` is replaced by its corresponding numerical value.

EXAMPLE:

```
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization = False)
sage: x = p.new_variable()
sage: p.set_objective(x[3] - x[5])
```

```
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 2.]])
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[3] + a2*x[5] <= a3)
sage: p.add_constraint(b1*x[3] + b2*x[5] <= b3)
sage: round(p.solve(),3)
-3.0
```

To return the optimal value of `x[3]`:

```
sage: round(p.get_values(x[3]),3)
-1.0
```

To get a dictionary identical to `x` containing optimal values for the corresponding variables

```
sage: x_sol = p.get_values(x)
sage: x_sol.keys()
[3, 5]
```

Obviously, it also works with variables of higher dimension:

```
sage: x_sol = p.get_values(x)
```

linear_constraints_parent()

Return the parent for all linear constraints

See `linear_functions` for more details.

EXAMPLES:

```
sage: p = SemidefiniteProgram()
sage: p.linear_constraints_parent()
Linear constraints over Real Double Field
```

linear_function(*x*)

Construct a new linear function

EXAMPLES:

```
sage: p = SemidefiniteProgram()
sage: p.linear_function({0:1})
x_0
```

linear_functions_parent()

Return the parent for all linear functions

EXAMPLES:

```
sage: p = SemidefiniteProgram()
sage: p.linear_functions_parent()
Linear functions over Real Double Field
```

new_variable(*name*='')

Returns an instance of `SDPVariable` associated to the current instance of `SemidefiniteProgram`.

A new variable `x` is defined by:

```
sage: p = SemidefiniteProgram()
sage: x = p.new_variable()
```

It behaves exactly as an usual dictionary would. It can use any key argument you may like, as `x[5]` or `x["b"]`, and has methods `items()` and `keys()`.

INPUT:

- `dim` – integer. Defines the dimension of the dictionary. If `x` has dimension 2, its fields will be of the form `x[key1][key2]`. Deprecated.
- `name` – string. Associates a name to the variable.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: x = p.new_variable()
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: p.add_constraint(a1*x[0]+a1*x[3] <= 0)
sage: p.show()
Maximization:

Constraints:
  constraint_0: [1.0 2.0][2.0 3.0]x_0 + [1.0 2.0][2.0 3.0]x_1 <=  [0 0][0 0]
Variables:
   x_0,  x_1
```

number_of_constraints()

Returns the number of constraints assigned so far.

EXAMPLE:

```
sage: p = SemidefiniteProgram(solver = "cvxopt")
sage: x = p.new_variable()
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 2.]])
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a3)
sage: p.add_constraint(b1*x[0] + b2*x[1] <= b3)
sage: p.add_constraint(b1*x[0] + a2*x[1] <= b3)
sage: p.number_of_constraints()
3
```

number_of_variables()

Returns the number of variables used so far.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: a = matrix([[1, 2.], [2., 3.]])
sage: p.add_constraint(a*p[0] - a*p[2] <=  2*a*p[4]  )
sage: p.number_of_variables()
3
```

set_objective(*obj*)

Sets the objective of the `SemidefiniteProgram`.

INPUT:

- `obj` – A semidefinite function to be optimized. (can also be set to `None` or `0` when just looking for a feasible solution)

EXAMPLE:

Let's solve the following semidefinite program:

```
Maximize:
  x + 5 * y
Constraints:
  [1,2][2,3]x + [1,1][1,1] y       <= [1,-1][-1,1]
Variables:
  x, y
```

This SDP can be solved as follows:

```
sage: p = SemidefiniteProgram(maximization=True)
sage: x = p.new_variable()
sage: p.set_objective(x[1] + 5*x[2])
sage: a1 = matrix([[1,2],[2,3]])
sage: a2 = matrix([[1,1],[1,1]])
sage: a3 = matrix([[1,-1],[-1,1]])
sage: p.add_constraint(a1*x[1]+a2*x[2] <= a3)
sage: round(p.solve(),5)
16.2
sage: p.set_objective(None)
sage: _ = p.solve()
```

set_problem_name (*name*)

Sets the name of the SemidefiniteProgram.

INPUT:

- name – A string representing the name of the SemidefiniteProgram.

EXAMPLE:

```
sage: p = SemidefiniteProgram()
sage: p.set_problem_name("Test program")
sage: p
Semidefinite Program "Test program" ( maximization, 0 variables, 0 constraints )
```

show ()

Displays the SemidefiniteProgram in a human-readable way.

EXAMPLES:

When constraints and variables have names

```
sage: p = SemidefiniteProgram()
sage: x = p.new_variable(name="hihi")
sage: a1 = matrix([[1,2],[2,3]])
sage: a2 = matrix([[2,3],[3,4]])
sage: a3 = matrix([[3,4],[4,5]])
sage: p.set_objective(x[0] - x[1])
sage: p.add_constraint(a1*x[0]+a2*x[1]<= a3)
sage: p.show()
Maximization:
  hihi[0] - hihi[1]
Constraints:
  constraint_0: [1.0 2.0][2.0 3.0]hihi[0] + [2.0 3.0][3.0 4.0]hihi[1] <=  [3.0 4.0][4.0 5.0]
Variables:
   hihi[0],  hihi[1]
```

solve (*objective_only=False*)

Solves the SemidefiniteProgram.

INPUT:

- `objective_only` – Boolean variable.
  - When set to `True`, only the objective function is returned.
  - When set to `False` (default), the optimal numerical values are stored (takes computational time).

OUTPUT:

The optimal value taken by the objective function.

TESTS:

The SDP from the header of this module:

```
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization = False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1])
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 2.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 1.]])
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a3)
sage: p.add_constraint(b1*x[0] + b2*x[1] <= b3)
sage: round(p.solve(),4)
-11.0
sage: x = p.get_values(x)
sage: round(x[0],4)
-8.0
sage: round(x[1],4)
3.0
```

solver_parameter(*name*, *value=None*)

Return or define a solver parameter

The solver parameters are by essence solver-specific, which means their meaning heavily depends on the solver used.

(If you do not know which solver you are using, then you are using cvxopt).

INPUT:

- `name` (string) – the parameter
- `value` – the parameter's value if it is to be defined, or `None` (default) to obtain its current value.

EXAMPLE:

```
sage: p.<x> = SemidefiniteProgram(solver = "cvxopt", maximization = False)
sage: p.solver_parameter("show_progress", True)
sage: p.solver_parameter("show_progress")
True
sage: p.set_objective(x[0] - x[1])
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 2.]])
sage: a3 = matrix([[5, 6.], [6., 7.]])
sage: b1 = matrix([[1, 1.], [1., 1.]])
sage: b2 = matrix([[2, 2.], [2., 1.]])
sage: b3 = matrix([[3, 3.], [3., 3.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a3)
sage: p.add_constraint(b1*x[0] + b2*x[1] <= b3)
```

```
sage: round(p.solve(),4)
     pcost       dcost       gap    pres   dres   k/t
 0:  1...
...
Optimal solution found.
-11.0
```

sum(*L*)

Efficiently computes the sum of a sequence of `LinearFunction` elements

INPUT:

- `L` – list of `LinearFunction` instances.

Note: The use of the regular `sum` function is not recommended as it is much less efficient than this one

EXAMPLES:

```
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
```

The following command:

```
sage: s = p.sum([v[i] for i in xrange(90)])
```

is much more efficient than:

```
sage: s = sum([v[i] for i in xrange(90)])
```

CHAPTER

FOUR

LINEAR FUNCTIONS AND CONSTRAINTS

This module implements linear functions (see `LinearFunction`) in formal variables and chained (in)equalities between them (see `LinearConstraint`). By convention, these are always written as either equalities or less-or-equal. For example:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: f = 1 + x[1] + 2*x[2];  f     #  a linear function
1 + x_0 + 2*x_1
sage: type(f)
<type 'sage.numerical.linear_functions.LinearFunction'>

sage: c = (0 <= f);  c    # a constraint
0 <= 1 + x_0 + 2*x_1
sage: type(c)
<type 'sage.numerical.linear_functions.LinearConstraint'>
```

Note that you can use this module without any reference to linear programming, it only implements linear functions over a base ring and constraints. However, for ease of demonstration we will always construct them out of linear programs (see `mip`).

Constraints can be equations or (non-strict) inequalities. They can be chained:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: x[0] == x[1] == x[2] == x[3]
x_0 == x_1 == x_2 == x_3

sage: ieq_01234 = x[0] <= x[1] <= x[2] <= x[3] <= x[4]
sage: ieq_01234
x_0 <= x_1 <= x_2 <= x_3 <= x_4
```

If necessary, the direction of inequality is flipped to always write inequalities as less or equal:

```
sage: x[5] >= ieq_01234
x_0 <= x_1 <= x_2 <= x_3 <= x_4 <= x_5

sage: (x[5]<=x[6]) >= ieq_01234
x_0 <= x_1 <= x_2 <= x_3 <= x_4 <= x_5 <= x_6
sage: (x[5]<=x[6]) <= ieq_01234
x_5 <= x_6 <= x_0 <= x_1 <= x_2 <= x_3 <= x_4
```

TESTS:

See trac ticket #12091

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: b[0] <= b[1] <= 2
x_0 <= x_1 <= 2
sage: list(b[0] <= b[1] <= 2)
[x_0, x_1, 2]
sage: 1 >= b[1] >= 2*b[0]
2*x_0 <= x_1 <= 1
sage: b[2] >= b[1] >= 2*b[0]
2*x_0 <= x_1 <= x_2
```

class `sage.numerical.linear_functions.`**`LinearConstraint`**

Bases: `sage.structure.element.Element`

A class to represent formal Linear Constraints.

A Linear Constraint being an inequality between two linear functions, this class lets the user write `LinearFunction1 <= LinearFunction2` to define the corresponding constraint, which can potentially involve several layers of such inequalities (`(A <= B <= C`), or even equalities like `A == B`.

Trivial constraints (meaning that they have only one term and no relation) are also allowed. They are required for the coercion system to work.

> **Warning:** This class has no reason to be instantiated by the user, and is meant to be used by instances of `MixedIntegerLinearProgram`.

INPUT:

- `parent` – the parent, a `LinearConstraintsParent_class`
- `terms` – a list/tuple/iterable of two or more linear functions (or things that can be converted into linear functions).
- `equality` – boolean (default: `False`). Whether the terms are the entries of a chained less-or-equal (`<=`) inequality or a chained equality.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: b[2]+2*b[3] <= b[8]-5
x_0 + 2*x_1 <= -5 + x_2
```

`equals`(*left*, *right*)

Compare `left` and `right`.

OUTPUT:

Boolean. Whether all terms of `left` and `right` are equal. Note that this is stronger than mathematical equivalence of the relations.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: (x[1] + 1 >= 2).equals(3/3 + 1*x[1] + 0*x[2] >= 8/4)
True
sage: (x[1] + 1 >= 2).equals(x[1] + 1-1 >= 1-1)
False
```

equations()

Iterate over the unchained(!) equations

OUTPUT:

An iterator over pairs (lhs, rhs) such that the individual equations are lhs == rhs.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: eqns = 1 == b[0] == b[2] == 3 == b[3];  eqns
1 == x_0 == x_1 == 3 == x_2

sage: for lhs, rhs in eqns.equations():
...       print str(lhs) + ' == ' + str(rhs)
1 == x_0
x_0 == x_1
x_1 == 3
3 == x_2
```

inequalities()

Iterate over the unchained(!) inequalities

OUTPUT:

An iterator over pairs (lhs, rhs) such that the individual equations are lhs <= rhs.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: ieq = 1 <= b[0] <= b[2] <= 3 <= b[3]; ieq
1 <= x_0 <= x_1 <= 3 <= x_2

sage: for lhs, rhs in ieq.inequalities():
...       print str(lhs) + ' <= ' + str(rhs)
1 <= x_0
x_0 <= x_1
x_1 <= 3
3 <= x_2
```

is_equation()

Whether the constraint is a chained equation

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: (b[0] == b[1]).is_equation()
True
sage: (b[0] <= b[1]).is_equation()
False
```

is_less_or_equal()

Whether the constraint is a chained less-or_equal inequality

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
sage: (b[0] == b[1]).is_less_or_equal()
False
sage: (b[0] <= b[1]).is_less_or_equal()
True
```

is_trivial()

Test whether the constraint is trivial.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: LC = p.linear_constraints_parent()
sage: ieq = LC(1,2);  ieq
1 <= 2
sage: ieq.is_trivial()
False

sage: ieq = LC(1);  ieq
trivial constraint starting with 1
sage: ieq.is_trivial()
True
```

sage.numerical.linear_functions.**LinearConstraintsParent**(*linear_functions_parent*)

Return the parent for linear functions over base_ring.

The output is cached, so only a single parent is ever constructed for a given base ring.

INPUT:

- linear_functions_parent – a LinearFunctionsParent_class. The type of linear functions that the constraints are made out of.

OUTPUT:

The parent of the linear constraints with the given linear functions.

EXAMPLES:

```
sage: from sage.numerical.linear_functions import          ...       LinearFunctionsParent, I
sage: LF = LinearFunctionsParent(QQ)
sage: LinearConstraintsParent(LF)
Linear constraints over Rational Field
```

class sage.numerical.linear_functions.**LinearConstraintsParent_class**

Bases: sage.structure.parent.Parent

Parent for LinearConstraint

> **Warning:** This class has no reason to be instantiated by the user, and is meant to be used by instances of MixedIntegerLinearProgram. Also, use the LinearConstraintsParent() factory function.

INPUT/OUTPUT:

See LinearFunctionsParent()

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: LC = p.linear_constraints_parent();  LC
Linear constraints over Real Double Field
sage: from sage.numerical.linear_functions import LinearConstraintsParent
sage: LinearConstraintsParent(p.linear_functions_parent()) is LC
True
```

linear_functions_parent()

Return the parent for the linear functions

EXAMPLES:

```
sage: LC = MixedIntegerLinearProgram().linear_constraints_parent()
sage: LC.linear_functions_parent()
Linear functions over Real Double Field
```

class sage.numerical.linear_functions.**LinearFunction**

Bases: sage.structure.element.ModuleElement

An elementary algebra to represent symbolic linear functions.

> **Warning:** You should never instantiate LinearFunction manually. Use the element constructor in the parent instead. For convenience, you can also call the MixedIntegerLinearProgram instance directly.

EXAMPLES:

For example, do this:

```
sage: p = MixedIntegerLinearProgram()
sage: p({0 : 1, 3 : -8})
x_0 - 8*x_3
```

or this:

```
sage: parent = p.linear_functions_parent()
sage: parent({0 : 1, 3 : -8})
x_0 - 8*x_3
```

instead of this:

```
sage: from sage.numerical.linear_functions import LinearFunction
sage: LinearFunction(p.linear_functions_parent(), {0 : 1, 3 : -8})
x_0 - 8*x_3
```

coefficient(x)

Return one of the the coefficients.

INPUT:

- x – a linear variable or an integer. If an integer i is passed, then x_i is used as linear variable.

OUTPUT:

A base ring element. The coefficient of x in the linear function. Pass -1 for the constant term.

EXAMPLE:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: lf = -8 * b[3] + b[0] - 5;  lf
-5 - 8*x_0 + x_1
sage: lf.coefficient(b[3])
```

```
-8.0
sage: lf.coefficient(0)      # x_0 is b[3]
-8.0
sage: lf.coefficient(4)
0.0
sage: lf.coefficient(-1)
-5.0
```

TESTS:

```
sage: lf.coefficient(b[3] + b[4])
Traceback (most recent call last):
...
ValueError: x is a sum, must be a single variable
sage: lf.coefficient(2*b[3])
Traceback (most recent call last):
...
ValueError: x must have a unit coefficient
sage: mip.<q> = MixedIntegerLinearProgram(solver='ppl')
sage: lf.coefficient(q[0])
Traceback (most recent call last):
...
ValueError: x is from a different linear functions module
```

dict()

Return the dictionary corresponding to the Linear Function.

OUTPUT:

The linear function is represented as a dictionary. The value are the coefficient of the variable represented by the keys (which are integers). The key `-1` corresponds to the constant term.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram()
sage: lf = p({0 : 1, 3 : -8})
sage: lf.dict()
{0: 1.0, 3: -8.0}
```

equals(*left*, *right*)

Logically compare `left` and `right`.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: (x[1] + 1).equals(3/3 + 1*x[1] + 0*x[2])
True
```

is_zero()

Test whether `self` is zero.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: (x[1] - x[1] + 0*x[2]).is_zero()
True
```

iteritems()

Iterate over the index, coefficient pairs

OUTPUT:

An iterator over the `(key, coefficient)` pairs. The keys are integers indexing the variables. The key `-1` corresponds to the constant term.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(solver = 'ppl')
sage: x = p.new_variable()
sage: f = 0.5 + 3/2*x[1] + 0.6*x[3]
sage: for id, coeff in f.iteritems():
...       print 'id =', id, '  coeff =', coeff
id = 0   coeff = 3/2
id = 1   coeff = 3/5
id = -1   coeff = 1/2
```

`sage.numerical.linear_functions.`**LinearFunctionsParent**(*base_ring*)

Return the parent for linear functions over `base_ring`.

The output is cached, so only a single parent is ever constructed for a given base ring.

INPUT:

- `base_ring` – a ring. The coefficient ring for the linear functions.

OUTPUT:

The parent of the linear functions over `base_ring`.

EXAMPLES:

```
sage: from sage.numerical.linear_functions import LinearFunctionsParent
sage: LinearFunctionsParent(QQ)
Linear functions over Rational Field
```

class `sage.numerical.linear_functions.`**LinearFunctionsParent_class**

Bases: `sage.structure.parent.Parent`

The parent for all linear functions over a fixed base ring.

Warning: You should use `LinearFunctionsParent()` to construct instances of this class.

INPUT/OUTPUT:

See `LinearFunctionsParent()`

EXAMPLES:

```
sage: from sage.numerical.linear_functions import LinearFunctionsParent_class
sage: LinearFunctionsParent_class
<type 'sage.numerical.linear_functions.LinearFunctionsParent_class'>
```

gen(*i*)

Return the linear variable x_i.

INPUT:

- i – non-negative integer.

OUTPUT:

The linear function x_i.

EXAMPLES:

```
sage: LF = MixedIntegerLinearProgram().linear_functions_parent()
sage: LF.gen(23)
x_23
```

set_multiplication_symbol(*symbol='*'*)

Set the multiplication symbol when pretty-printing linear functions.

INPUT:

- symbol – string, default: '*'. The multiplication symbol to be used.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: f = -1-2*x[0]-3*x[1]
sage: LF = f.parent()
sage: LF._get_multiplication_symbol()
'*'
sage: f
-1 - 2*x_0 - 3*x_1
sage: LF.set_multiplication_symbol(' ')
sage: f
-1 - 2 x_0 - 3 x_1
sage: LF.set_multiplication_symbol()
sage: f
-1 - 2*x_0 - 3*x_1
```

tensor(*free_module*)

Return the tensor product with free_module.

INPUT:

- free_module – vector space or matrix space over the same base ring.

OUTPUT:

Instance of sage.numerical.linear_tensor.LinearTensorParent_class.

EXAMPLES:

```
sage: LF = MixedIntegerLinearProgram().linear_functions_parent()
sage: LF.tensor(RDF^3)
Tensor product of Vector space of dimension 3 over Real Double Field
and Linear functions over Real Double Field
sage: LF.tensor(QQ^2)
Traceback (most recent call last):
...
ValueError: base rings must match
```

sage.numerical.linear_functions.**is_LinearConstraint**(*x*)

Test whether x is a linear constraint

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: ieq = (x[0] <= x[1])
sage: from sage.numerical.linear_functions import is_LinearConstraint
sage: is_LinearConstraint(ieq)
True
sage: is_LinearConstraint('a string')
False
```

sage.numerical.linear_functions.**is_LinearFunction**(*x*)

Test whether x is a linear function

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
sage: from sage.numerical.linear_functions import is_LinearFunction
sage: is_LinearFunction(x[0] - 2*x[2])
True
sage: is_LinearFunction('a string')
False
```

CHAPTER

FIVE

MATRIX/VECTOR-VALUED LINEAR FUNCTIONS: PARENTS

In Sage, matrices assume that the base is a ring. Hence, we cannot construct matrices whose entries are linear functions in Sage. Really, they should be thought of as the tensor product of the R-module of linear functions and the R-module of vector/matrix spaces (R is QQ or RDF for our purposes).

You should not construct any tensor products by calling the parent directly. This is also why none of the classes are imported in the global namespace. The come into play whenever you have vector or matrix MIP linear expressions/constraints. The intended way to construct them is implicitly by acting with vectors or matrices on linear functions. For example:

```
sage: mip.<x> = MixedIntegerLinearProgram('ppl')    # base ring is QQ
sage: 3 + x[0] + 2*x[1]             # a linear function
3 + x_0 + 2*x_1
sage: x[0] * vector([3,4]) + 1      # vector linear function
(1, 1) + (3, 4)*x_0
sage: x[0] * matrix([[3,1],[4,0]]) + 1    # matrix linear function
[1 + 3*x_0 x_0]
[4*x_0     1  ]
```

Internally, all linear functions are stored as a dictionary whose

- keys are the index of the linear variable (and -1 for the constant term)
- values are the coefficient of that variable. That is, a number for linear functions, a vector for vector-valued functions, etc.

The entire dictionary can be accessed with the `dict()` method. For convenience, you can also retrieve a single coefficient with `coefficient()`. For example:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: f_scalar = (3 + b[7] + 2*b[9]);  f_scalar
3 + x_0 + 2*x_1
sage: f_scalar.dict()
{-1: 3.0, 0: 1.0, 1: 2.0}
sage: f_scalar.dict()[1]
2.0
sage: f_scalar.coefficient(b[9])
2.0
sage: f_scalar.coefficient(1)
2.0

sage: f_vector = b[7] * vector([3,4]) + 1;  f_vector
(1.0, 1.0) + (3.0, 4.0)*x_0
sage: f_vector.coefficient(-1)
(1.0, 1.0)
sage: f_vector.coefficient(b[7])
```

```
(3.0, 4.0)
sage: f_vector.coefficient(0)
(3.0, 4.0)
sage: f_vector.coefficient(1)
(0.0, 0.0)

sage: f_matrix = b[7] * matrix([[0,1], [2,0]]) + b[9] - 3;  f_matrix
[-3 + x_1 x_0      ]
[2*x_0    -3 + x_1]
sage: f_matrix.coefficient(-1)
[-3.0  0.0]
[ 0.0 -3.0]
sage: f_matrix.coefficient(0)
[0.0 1.0]
[2.0 0.0]
sage: f_matrix.coefficient(1)
[1.0 0.0]
[0.0 1.0]
```

Just like `sage.numerical.linear_functions`, (in)equalities become symbolic inequalities. See `linear_tensor_constraints` for detais.

Note: For brevity, we just use `LinearTensor` in class names. It is understood that this refers to the above tensor product construction.

`sage.numerical.linear_tensor.`**`LinearTensorParent`**(*free_module_parent*, *linear_functions_parent*)

Return the parent for the tensor product over the common `base_ring`.

The output is cached, so only a single parent is ever constructed for a given base ring.

INPUT:

- `free_module_parent` – module. A free module, like vector or matrix space.
- `linear_functions_parent` – linear functions. The linear functions parent.

OUTPUT:

The parent of the tensor product of a free module and linear functions over a common base ring.

EXAMPLES:

```
sage: from sage.numerical.linear_functions import LinearFunctionsParent
sage: from sage.numerical.linear_tensor import LinearTensorParent
sage: LinearTensorParent(QQ^3, LinearFunctionsParent(QQ))
Tensor product of Vector space of dimension 3 over Rational Field and Linear functions over Rati

sage: LinearTensorParent(ZZ^3, LinearFunctionsParent(QQ))
Traceback (most recent call last):
...
ValueError: base rings must match
```

class `sage.numerical.linear_tensor.`**`LinearTensorParent_class`**(*free_module*, *linear_functions*)

Bases: `sage.structure.parent.Parent`

The parent for all linear functions over a fixed base ring.

Warning: You should use `LinearTensorParent()` to construct instances of this class.

INPUT/OUTPUT:

See `LinearTensorParent()`

EXAMPLES:

```
sage: from sage.numerical.linear_tensor import LinearTensorParent_class
sage: LinearTensorParent_class
<class 'sage.numerical.linear_tensor.LinearTensorParent_class'>
```

Element
alias of `LinearTensor`

free_module()
Return the linear functions.

See also `free_module()`.

OUTPUT:

Parent of the linear functions, one of the factors in the tensor product construction.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: lt = x[0] * vector(RDF, [1,2])
sage: lt.parent().free_module()
Vector space of dimension 2 over Real Double Field
sage: lt.parent().free_module() is vector(RDF, [1,2]).parent()
True
```

is_matrix_space()
Return whether the free module is a matrix space.

OUTPUT:

Boolean. Whether the `free_module()` factor in the tensor product is a matrix space.

EXAMPLES:

```
sage: mip = MixedIntegerLinearProgram()
sage: LF = mip.linear_functions_parent()
sage: LF.tensor(RDF^2).is_matrix_space()
False
sage: LF.tensor(RDF^(2,2)).is_matrix_space()
True
```

is_vector_space()
Return whether the free module is a vector space.

OUTPUT:

Boolean. Whether the `free_module()` factor in the tensor product is a vector space.

EXAMPLES:

```
sage: mip = MixedIntegerLinearProgram()
sage: LF = mip.linear_functions_parent()
sage: LF.tensor(RDF^2).is_vector_space()
True
sage: LF.tensor(RDF^(2,2)).is_vector_space()
False
```

linear_functions()

Return the linear functions.

See also `free_module()`.

OUTPUT:

Parent of the linear functions, one of the factors in the tensor product construction.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: lt = x[0] * vector([1,2])
sage: lt.parent().linear_functions()
Linear functions over Real Double Field
sage: lt.parent().linear_functions() is mip.linear_functions_parent()
True
```

sage.numerical.linear_tensor.**is_LinearTensor**(*x*)

Test whether x is a tensor product of linear functions with a free module.

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable(nonnegative=False)
sage: from sage.numerical.linear_tensor import is_LinearTensor
sage: is_LinearTensor(x[0] - 2*x[2])
False
sage: is_LinearTensor('a string')
False
```

CHAPTER

SIX

MATRIX/VECTOR-VALUED LINEAR FUNCTIONS: ELEMENTS

Here is an example of a linear function tensored with a vector space:

```
sage: mip.<x> = MixedIntegerLinearProgram('ppl')   # base ring is QQ
sage: lt = x[0] * vector([3,4]) + 1;   lt
(1, 1) + (3, 4)*x_0
sage: type(lt)
<type 'sage.numerical.linear_tensor_element.LinearTensor'>
```

class `sage.numerical.linear_tensor_element.`**`LinearTensor`**

Bases: `sage.structure.element.ModuleElement`

A linear function tensored with a free module

> **Warning:** You should never instantiate `LinearTensor` manually. Use the element constructor in the parent instead.

EXAMPLES:

```
sage: parent = MixedIntegerLinearProgram().linear_functions_parent().tensor(RDF^2)
sage: parent({0: [1,2], 3: [-7,-8]})
(1.0, 2.0)*x_0 + (-7.0, -8.0)*x_3
```

`coefficient`(*x*)

Return one of the the coefficients.

INPUT:

- `x` – a linear variable or an integer. If an integer i is passed, then x_i is used as linear variable. Pass `-1` for the constant term.

OUTPUT:

A constant, that is, an element of the free module factor. The coefficient of `x` in the linear function.

EXAMPLE:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: lt = vector([1,2]) * b[3] + vector([4,5]) * b[0] - 5;  lt
(-5.0, -5.0) + (1.0, 2.0)*x_0 + (4.0, 5.0)*x_1
sage: lt.coefficient(b[3])
(1.0, 2.0)
sage: lt.coefficient(0)       # x_0 is b[3]
(1.0, 2.0)
sage: lt.coefficient(4)
(0.0, 0.0)
sage: lt.coefficient(-1)
(-5.0, -5.0)
```

TESTS:

```
sage: lt.coefficient(b[3] + b[4])
Traceback (most recent call last):
...
ValueError: x is a sum, must be a single variable
sage: lt.coefficient(2*b[3])
Traceback (most recent call last):
...
ValueError: x must have a unit coefficient
sage: mip.<q> = MixedIntegerLinearProgram(solver='ppl')
sage: lt.coefficient(q[0])
Traceback (most recent call last):
...
ValueError: x is from a different linear functions module
```

dict()

Return the dictionary corresponding to the tensor product.

OUTPUT:

The linear function tensor product is represented as a dictionary. The value are the coefficient (free module elements) of the variable represented by the keys (which are integers). The key `-1` corresponds to the constant term.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram().linear_functions_parent().tensor(RDF^2)
sage: lt = p({0:[1,2], 3:[4,5]})
sage: lt.dict()
{0: (1.0, 2.0), 3: (4.0, 5.0)}
```

CHAPTER

SEVEN

CONSTRAINTS ON LINEAR FUNCTIONS TENSORED WITH A FREE MODULE

Here is an example of a vector-valued linear function:

```
sage: mip.<x> = MixedIntegerLinearProgram('ppl')    # base ring is QQ
sage: x[0] * vector([3,4]) + 1      # vector linear function
(1, 1) + (3, 4)*x_0
```

Just like `linear_functions`, (in)equalities become symbolic inequalities:

```
sage: 3 + x[0] + 2*x[1] <= 10
3 + x_0 + 2*x_1 <= 10
sage: x[0] * vector([3,4]) + 1 <= 10
(1, 1) + (3, 4)*x_0 <= (10, 10)
sage: x[0] * matrix([[0,0,1],[0,1,0],[1,0,0]]) + x[1] * identity_matrix(3) >= 0
[0 0 0]    [x_1 0         x_0]
[0 0 0] <= [0   x_0 + x_1 0  ]
[0 0 0]    [x_0 0         x_1]
```

class `sage.numerical.linear_tensor_constraints.`**`LinearTensorConstraint`**(*parent*, *lhs*, *rhs*, *equality*)

Bases: `sage.structure.element.Element`

Formal constraint involving two module-valued linear functions.

Note: In the code, we use "linear tensor" as abbreviation for the tensor product (over the common base ring) of a `linear function` and a free module like a vector/matrix space.

Warning: This class has no reason to be instantiated by the user, and is meant to be used by instances of `MixedIntegerLinearProgram`.

INPUT:

- `parent` – the parent, a `LinearTensorConstraintsParent_class`
- `lhs`, `rhs` – two `sage.numerical.linear_tensor_element.LinearTensor`. The left and right hand side of the constraint (in)equality.
- `equality` – boolean (default: `False`). Whether the constraint is an equality. If `False`, it is a <= inequality.

EXAMPLE:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: (b[2]+2*b[3]) * vector([1,2]) <= b[8] * vector([2,3]) - 5
(1.0, 2.0)*x_0 + (2.0, 4.0)*x_1 <= (-5.0, -5.0) + (2.0, 3.0)*x_2
```

is_equation()

Whether the constraint is a chained equation

OUTPUT:

Boolean.

EXAMPLES:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: (b[0] * vector([1,2]) == 0).is_equation()
True
sage: (b[0] * vector([1,2]) >= 0).is_equation()
False
```

is_less_or_equal()

Whether the constraint is a chained less-or_equal inequality

OUTPUT:

Boolean.

EXAMPLES:

```
sage: mip.<b> = MixedIntegerLinearProgram()
sage: (b[0] * vector([1,2]) == 0).is_less_or_equal()
False
sage: (b[0] * vector([1,2]) >= 0).is_less_or_equal()
True
```

lhs()

Return the left side of the (in)equality.

OUTPUT:

Instance of `sage.numerical.linear_tensor_element.LinearTensor`. A linear function valued in a free module.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: (x[0] * vector([1,2]) == 0).lhs()
(1.0, 2.0)*x_0
```

rhs()

Return the right side of the (in)equality.

OUTPUT:

Instance of `sage.numerical.linear_tensor_element.LinearTensor`. A linear function valued in a free module.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: (x[0] * vector([1,2]) == 0).rhs()
(0.0, 0.0)
```

sage.numerical.linear_tensor_constraints.**LinearTensorConstraintsParent**(*linear_functions_parent*)

Return the parent for linear functions over base_ring.

The output is cached, so only a single parent is ever constructed for a given base ring.

INPUT:

- linear_functions_parent – a LinearFunctionsParent_class. The type of linear functions that the constraints are made out of.

OUTPUT:

The parent of the linear constraints with the given linear functions.

EXAMPLES:

```
sage: from sage.numerical.linear_functions import LinearFunctionsParent
sage: from sage.numerical.linear_tensor import LinearTensorParent
sage: from sage.numerical.linear_tensor_constraints import         ...       LinearTensorCon
sage: LF = LinearFunctionsParent(QQ)
sage: LT = LinearTensorParent(QQ^2, LF)
sage: LinearTensorConstraintsParent(LT)
Linear constraints in the tensor product of Vector space of dimension 2
over Rational Field and Linear functions over Rational Field
```

class sage.numerical.linear_tensor_constraints.**LinearTensorConstraintsParent_class**(*linear_tensor_*

Bases: sage.structure.parent.Parent

Parent for LinearTensorConstraint

> **Warning:** This class has no reason to be instantiated by the user, and is meant to be used by instances of MixedIntegerLinearProgram. Also, use the LinearTensorConstraintsParent() factory function.

INPUT/OUTPUT:

See LinearTensorConstraintsParent()

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: LT = p.linear_functions_parent().tensor(RDF^2);  LT
Tensor product of Vector space of dimension 2 over Real Double
Field and Linear functions over Real Double Field
sage: from sage.numerical.linear_tensor_constraints import LinearTensorConstraintsParent
sage: LTC = LinearTensorConstraintsParent(LT);  LTC
Linear constraints in the tensor product of Vector space of
dimension 2 over Real Double Field and Linear functions over
Real Double Field
sage: type(LTC)
<class 'sage.numerical.linear_tensor_constraints.LinearTensorConstraintsParent_class'>
```

Element

alias of LinearTensorConstraint

linear_functions()

Return the parent for the linear functions

OUTPUT:

Instance of sage.numerical.linear_functions.LinearFunctionsParent_class.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: ieq = (x[0] * vector([1,2]) >= 0)
sage: ieq.parent().linear_functions()
Linear functions over Real Double Field
```

linear_tensors()

Return the parent for the linear functions

OUTPUT:

Instance of `sage.numerical.linear_tensor.LinearTensorParent_class`.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: ieq = (x[0] * vector([1,2]) >= 0)
sage: ieq.parent().linear_tensors()
Tensor product of Vector space of dimension 2 over Real Double
Field and Linear functions over Real Double Field
```

sage.numerical.linear_tensor_constraints.**is_LinearTensorConstraint**(*x*)

Test whether `x` is a constraint on module-valued linear functions.

INPUT:

- `x` – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: mip.<x> = MixedIntegerLinearProgram()
sage: vector_ieq = (x[0] * vector([1,2]) <= x[1] * vector([2,3]))
sage: from sage.numerical.linear_tensor_constraints import is_LinearTensorConstraint
sage: is_LinearTensorConstraint(vector_ieq)
True
sage: is_LinearTensorConstraint('a string')
False
```

CHAPTER

EIGHT

NUMERICAL ROOT FINDING AND OPTIMIZATION

AUTHOR:

- William Stein (2007): initial version
- Nathann Cohen (2008) : Bin Packing

8.1 Functions and Methods

`sage.numerical.optimize.binpacking`(*items*, *maximum=1*, *k=None*)

Solves the bin packing problem.

The Bin Packing problem is the following :

Given a list of items of weights p_i and a real value K, what is the least number of bins such that all the items can be put in the bins, while keeping sure that each bin contains a weight of at most K ?

For more informations : http://en.wikipedia.org/wiki/Bin_packing_problem

Two version of this problem are solved by this algorithm :

- Is it possible to put the given items in L bins ?
- What is the assignment of items using the least number of bins with the given list of items ?

INPUT:

- `items` – A list of real values (the items' weight)
- `maximum` – The maximal size of a bin
- `k` – Number of bins
  - When set to an integer value, the function returns a partition of the items into k bins if possible, and raises an exception otherwise.
  - When set to `None`, the function returns a partition of the items using the least number possible of bins.

OUTPUT:

A list of lists, each member corresponding to a box and containing the list of the weights inside it. If there is no solution, an exception is raised (this can only happen when `k` is specified or if `maximum` is less that the size of one item).

EXAMPLES:

Trying to find the minimum amount of boxes for 5 items of weights $1/5, 1/4, 2/3, 3/4, 5/7$:

```
sage: from sage.numerical.optimize import binpacking
sage: values = [1/5, 1/3, 2/3, 3/4, 5/7]
sage: bins = binpacking(values)
sage: len(bins)
3
```

Checking the bins are of correct size

```
sage: all([ sum(b)<= 1 for b in bins ])
True
```

Checking every item is in a bin

```
sage: b1, b2, b3 = bins
sage: all([ (v in b1 or v in b2 or v in b3) for v in values ])
True
```

One way to use only three boxes (which is best possible) is to put $1/5 + 3/4$ together in a box, $1/3 + 2/3$ in another, and $5/7$ by itself in the third one.

Of course, we can also check that there is no solution using only two boxes

```
sage: from sage.numerical.optimize import binpacking
sage: binpacking([0.2,0.3,0.8,0.9], k=2)
Traceback (most recent call last):
...
ValueError: This problem has no solution !
```

`sage.numerical.optimize.`**`find_fit`**`(`*data*, *model*, *initial_guess=None*, *parameters=None*, *variables=None*, *solution_dict=False*`)`

Finds numerical estimates for the parameters of the function model to give a best fit to data.

INPUT:

- `data` – A two dimensional table of floating point numbers of the form $[[x_{1,1}, x_{1,2}, \ldots, x_{1,k}, f_1], [x_{2,1}, x_{2,2}, \ldots, x_{2,k}, f_2], \ldots, [x_{n,1}, x_{n,2}, \ldots, x_{n,k}, f_n]]$ given as either a list of lists, matrix, or numpy array.
- `model` – Either a symbolic expression, symbolic function, or a Python function. `model` has to be a function of the variables $(x_1, x_2, \ldots, x_k)$ and free parameters $(a_1, a_2, \ldots, a_l)$.
- `initial_guess` – (default: `None`) Initial estimate for the parameters $(a_1, a_2, \ldots, a_l)$, given as either a list, tuple, vector or numpy array. If `None`, the default estimate for each parameter is 1.
- `parameters` – (default: `None`) A list of the parameters $(a_1, a_2, \ldots, a_l)$. If model is a symbolic function it is ignored, and the free parameters of the symbolic function are used.
- `variables` – (default: `None`) A list of the variables $(x_1, x_2, \ldots, x_k)$. If model is a symbolic function it is ignored, and the variables of the symbolic function are used.
- `solution_dict` – (default: `False`) if `True`, return the solution as a dictionary rather than an equation.

EXAMPLES:

First we create some data points of a sine function with some random perturbations:

```
sage: data = [(i, 1.2 * sin(0.5*i-0.2) + 0.1 * normalvariate(0, 1)) for i in xsrange(0, 4*pi, 0.
sage: var('a, b, c, x')
(a, b, c, x)
```

We define a function with free parameters a, b and c:

```
sage: model(x) = a * sin(b * x - c)
```

We search for the parameters that give the best fit to the data:

```
sage: find_fit(data, model)
[a == 1.21..., b == 0.49..., c == 0.19...]
```

We can also use a Python function for the model:

```
sage: def f(x, a, b, c): return a * sin(b * x - c)
sage: fit = find_fit(data, f, parameters = [a, b, c], variables = [x], solution_dict = True)
sage: fit[a], fit[b], fit[c]
(1.21..., 0.49..., 0.19...)
```

We search for a formula for the n-th prime number:

```
sage: dataprime = [(i, nth_prime(i)) for i in xrange(1, 5000, 100)]
sage: find_fit(dataprime, a * x * log(b * x), parameters = [a, b], variables = [x])
[a == 1.11..., b == 1.24...]
```

ALGORITHM:

Uses `scipy.optimize.leastsq` which in turn uses MINPACK's lmdif and lmder algorithms.

`sage.numerical.optimize.`**`find_local_maximum`**(*f*, *a*, *b*, *tol=1.48e-08*, *maxfun=500*)

Numerically find a local maximum of the expression f on the interval $[a, b]$ (or $[b, a]$) along with the point at which the maximum is attained.

Note that this function only finds a *local* maximum, and not the global maximum on that interval – see the examples with `find_local_maximum()`.

See the documentation for `find_local_maximum()` for more details and possible workarounds for finding the global minimum on an interval.

EXAMPLES:

```
sage: f = lambda x: x*cos(x)
sage: find_local_maximum(f, 0, 5)
(0.561096338191..., 0.8603335890...)
sage: find_local_maximum(f, 0, 5, tol=0.1, maxfun=10)
(0.561090323458..., 0.857926501456...)
sage: find_local_maximum(8*e^(-x)*sin(x) - 1, 0, 7)
(1.579175535558..., 0.7853981...)
```

`sage.numerical.optimize.`**`find_local_minimum`**(*f*, *a*, *b*, *tol=1.48e-08*, *maxfun=500*)

Numerically find a local minimum of the expression `f` on the interval $[a, b]$ (or $[b, a]$) and the point at which it attains that minimum. Note that `f` must be a function of (at most) one variable.

Note that this function only finds a *local* minimum, and not the global minimum on that interval – see the examples below.

INPUT:

- `f` – a function of at most one variable.
- `a`, `b` – endpoints of interval on which to minimize self.
- `tol` – the convergence tolerance
- `maxfun` – maximum function evaluations

OUTPUT:

- `minval` – (float) the minimum value that self takes on in the interval $[a, b]$
- `x` – (float) the point at which self takes on the minimum value

EXAMPLES:

```
sage: f = lambda x: x*cos(x)
sage: find_local_minimum(f, 1, 5)
(-3.28837139559..., 3.4256184695...)
sage: find_local_minimum(f, 1, 5, tol=1e-3)
(-3.28837136189098..., 3.42575079030572...)
sage: find_local_minimum(f, 1, 5, tol=1e-2, maxfun=10)
(-3.28837084598..., 3.4250840220...)
sage: show(plot(f, 0, 20))
sage: find_local_minimum(f, 1, 15)
(-9.4772942594..., 9.5293344109...)
```

Only local minima are found; if you enlarge the interval, the returned minimum may be *larger*! See trac ticket #2607.

```
sage: f(x) = -x*sin(x^2)
sage: find_local_minimum(f, -2.5, -1)
(-2.182769784677722, -2.1945027498534686)
```

Enlarging the interval returns a larger minimum:

```
sage: find_local_minimum(f, -2.5, 2)
(-1.3076194129914434, 1.3552111405712108)
```

One work-around is to plot the function and grab the minimum from that, although the plotting code does not necessarily do careful numerics (observe the small number of decimal places that we actually test):

```
sage: plot(f, (x,-2.5, -1)).ymin()
-2.1827...
sage: plot(f, (x,-2.5, 2)).ymin()
-2.1827...
```

ALGORITHM:

Uses scipy.optimize.fminbound which uses Brent's method.

AUTHOR:

- William Stein (2007-12-07)

`sage.numerical.optimize.`**`find_root`**(*f*, *a*, *b*, *xtol=1e-12*, *rtol=4.5e-16*, *maxiter=100*, *full_output=False*)

Numerically find a root of `f` on the closed interval $[a, b]$ (or $[b, a]$) if possible, where `f` is a function in the one variable. Note: this function only works in fixed (machine) precision, it is not possible to get arbitrary precision approximations with it.

INPUT:

- `f` – a function of one variable or symbolic equality
- `a`, `b` – endpoints of the interval
- `xtol`, `rtol` – the routine converges when a root is known to lie within `xtol` of the value return. Should be ≥ 0. The routine modifies this to take into account the relative precision of doubles.
- `maxiter` – integer; if convergence is not achieved in `maxiter` iterations, an error is raised. Must be ≥ 0.

- `full_output` – bool (default: `False`), if `True`, also return object that contains information about convergence.

EXAMPLES:

An example involving an algebraic polynomial function:

```
sage: R.<x> = QQ[]
sage: f = (x+17)*(x-3)*(x-1/8)^3
sage: find_root(f, 0,4)
2.999999999999995
sage: find_root(f, 0,1)  # note -- precision of answer isn't very good on some machines.
0.124999...
sage: find_root(f, -20,-10)
-17.0
```

In Pomerance's book on primes he asserts that the famous Riemann Hypothesis is equivalent to the statement that the function $f(x)$ defined below is positive for all $x \geq 2.01$:

```
sage: def f(x):
...       return sqrt(x) * log(x) - abs(Li(x) - prime_pi(x))
```

We find where f equals, i.e., what value that is slightly smaller than 2.01 that could have been used in the formulation of the Riemann Hypothesis:

```
sage: find_root(f, 2, 4, rtol=0.0001)
2.0082...
```

This agrees with the plot:

```
sage: plot(f,2,2.01)
Graphics object consisting of 1 graphics primitive
```

`sage.numerical.optimize.`**`linear_program`**(*c*, *G*, *h*, *A=None*, *b=None*, *solver=None*)

Solves the dual linear programs:

- Minimize $c'x$ subject to $Gx + s = h$, $Ax = b$, and $s \geq 0$ where $'$ denotes transpose.
- Maximize $-h'z - b'y$ subject to $G'z + A'y + c = 0$ and $z \geq 0$.

INPUT:

- `c` – a vector
- `G` – a matrix
- `h` – a vector
- `A` – a matrix
- `b` — a vector
- **`solver` (optional) — solver to use. If None, the cvxopt's lp-solver** is used. If it is 'glpk', then glpk's solver is used.

These can be over any field that can be turned into a floating point number.

OUTPUT:

A dictionary `sol` with keys `x`, `s`, `y`, `z` corresponding to the variables above:

- `sol['x']` – the solution to the linear program
- `sol['s']` – the slack variables for the solution
- `sol['z']`, `sol['y']` – solutions to the dual program

EXAMPLES:

First, we minimize $-4x_1 - 5x_2$ subject to $2x_1 + x_2 \leq 3$, $x_1 + 2x_2 \leq 3$, $x_1 \geq 0$, and $x_2 \geq 0$:

```
sage: c=vector(RDF,[-4,-5])
sage: G=matrix(RDF,[[2,1],[1,2],[-1,0],[0,-1]])
sage: h=vector(RDF,[3,3,0,0])
sage: sol=linear_program(c,G,h)
sage: sol['x']
(0.999..., 1.000...)
```

Next, we maximize $x + y - 50$ subject to $50x + 24y \leq 2400$, $30x + 33y \leq 2100$, $x \geq 45$, and $y \geq 5$:

```
sage: v=vector([-1.0,-1.0,-1.0])
sage: m=matrix([[50.0,24.0,0.0],[30.0,33.0,0.0],[-1.0,0.0,0.0],[0.0,-1.0,0.0],[0.0,0.0,1.0],[0.0
sage: h=vector([2400.0,2100.0,-45.0,-5.0,1.0,-1.0])
sage: sol=linear_program(v,m,h)
sage: sol['x']
(45.000000..., 6.2499999..., 1.00000000...)
sage: sol=linear_program(v,m,h,solver='glpk')
GLPK Simplex Optimizer...
OPTIMAL LP SOLUTION FOUND
sage: sol['x']
(45.0..., 6.25..., 1.0...)
```

`sage.numerical.optimize.`**`minimize`**(*func*, *x0*, *gradient=None*, *hessian=None*, *algorithm='default'*, ***args*)

This function is an interface to a variety of algorithms for computing the minimum of a function of several variables.

INPUT:

- `func` – Either a symbolic function or a Python function whose argument is a tuple with n components
- `x0` – Initial point for finding minimum.
- `gradient` – Optional gradient function. This will be computed automatically for symbolic functions. For Python functions, it allows the use of algorithms requiring derivatives. It should accept a tuple of arguments and return a NumPy array containing the partial derivatives at that point.
- `hessian` – Optional hessian function. This will be computed automatically for symbolic functions. For Python functions, it allows the use of algorithms requiring derivatives. It should accept a tuple of arguments and return a NumPy array containing the second partial derivatives of the function.
- `algorithm` – String specifying algorithm to use. Options are `'default'` (for Python functions, the simplex method is the default) (for symbolic functions bfgs is the default):
  - `'simplex'`
  - `'powell'`
  - `'bfgs'` – (Broyden-Fletcher-Goldfarb-Shanno) requires `gradient`
  - `'cg'` – (conjugate-gradient) requires gradient
  - `'ncg'` – (newton-conjugate gradient) requires gradient and hessian

EXAMPLES:

```
sage: vars=var('x y z')
sage: f=100*(y-x^2)^2+(1-x)^2+100*(z-y^2)^2+(1-y)^2
sage: minimize(f,[.1,.3,.4],disp=0)
(1.00..., 1.00..., 1.00...)
```

```
sage: minimize(f,[.1,.3,.4],algorithm="ncg",disp=0)
(0.9999999..., 0.999999..., 0.999999...)
```

Same example with just Python functions:

```
sage: def rosen(x): # The Rosenbrock function
...       return sum(100.0r*(x[1r:]-x[:-1r]**2.0r)**2.0r + (1r-x[:-1r])**2.0r)
sage: minimize(rosen,[.1,.3,.4],disp=0)
(1.00..., 1.00..., 1.00...)
```

Same example with a pure Python function and a Python function to compute the gradient:

```
sage: def rosen(x): # The Rosenbrock function
...       return sum(100.0r*(x[1r:]-x[:-1r]**2.0r)**2.0r + (1r-x[:-1r])**2.0r)
sage: import numpy
sage: from numpy import zeros
sage: def rosen_der(x):
...       xm = x[1r:-1r]
...       xm_m1 = x[:-2r]
...       xm_p1 = x[2r:]
...       der = zeros(x.shape,dtype=float)
...       der[1r:-1r] = 200r*(xm-xm_m1**2r) - 400r*(xm_p1 - xm**2r)*xm - 2r*(1r-xm)
...       der[0] = -400r*x[0r]*(x[1r]-x[0r]**2r) - 2r*(1r-x[0])
...       der[-1] = 200r*(x[-1r]-x[-2r]**2r)
...       return der
sage: minimize(rosen,[.1,.3,.4],gradient=rosen_der,algorithm="bfgs",disp=0)
(1.00...,  1.00..., 1.00...)
```

`sage.numerical.optimize.`**`minimize_constrained`**(*func*, *cons*, *x0*, *gradient=None*, *algorithm='default'*, ***args*)

Minimize a function with constraints.

INPUT:

- `func` – Either a symbolic function, or a Python function whose argument is a tuple with n components
- `cons` – constraints. This should be either a function or list of functions that must be positive. Alternatively, the constraints can be specified as a list of intervals that define the region we are minimizing in. If the constraints are specified as functions, the functions should be functions of a tuple with n components (assuming n variables). If the constraints are specified as a list of intervals and there are no constraints for a given variable, that component can be (`None`, `None`).
- `x0` – Initial point for finding minimum
- `algorithm` – Optional, specify the algorithm to use:
  - `'default'` – default choices
  - `'l-bfgs-b'` – only effective if you specify bound constraints. See [ZBN97].
- `gradient` – Optional gradient function. This will be computed automatically for symbolic functions. This is only used when the constraints are specified as a list of intervals.

EXAMPLES:

Let us maximize $x+y-50$ subject to the following constraints: $50x+24y \leq 2400$, $30x+33y \leq 2100$, $x \geq 45$, and $y \geq 5$:

```
sage: y = var('y')
sage: f = lambda p: -p[0]-p[1]+50
sage: c_1 = lambda p: p[0]-45
sage: c_2 = lambda p: p[1]-5
```

```
sage: c_3 = lambda p: -50*p[0]-24*p[1]+2400
sage: c_4 = lambda p: -30*p[0]-33*p[1]+2100
sage: a = minimize_constrained(f,[c_1,c_2,c_3,c_4],[2,3])
sage: a
(45.0, 6.25...)
```

Let's find a minimum of $\sin(xy)$:

```
sage: x,y = var('x y')
sage: f = sin(x*y)
sage: minimize_constrained(f, [(None,None),(4,10)],[5,5])
(4.8..., 4.8...)
```

Check, if L-BFGS-B finds the same minimum:

```
sage: minimize_constrained(f, [(None,None),(4,10)],[5,5], algorithm='l-bfgs-b')
(4.7..., 4.9...)
```

Rosenbrock function, [http://en.wikipedia.org/wiki/Rosenbrock_function]:

```
sage: from scipy.optimize import rosen, rosen_der
sage: minimize_constrained(rosen, [(-50,-10),(5,10)],[1,1],gradient=rosen_der,algorithm='l-bfgs-
(-10.0, 10.0)
sage: minimize_constrained(rosen, [(-50,-10),(5,10)],[1,1],algorithm='l-bfgs-b')
(-10.0, 10.0)
```

REFERENCES:

CHAPTER

NINE

INTERACTIVE SIMPLEX METHOD

This module, meant for **educational purposes only**, supports learning and exploring of the simplex method.

Do you want to solve Linear Programs efficiently? use `MixedIntegerLinearProgram` instead.

The methods implemented here allow solving Linear Programming Problems (LPPs) in a number of ways, may require explicit (and correct!) description of steps and are likely to be much slower than "regular" LP solvers. If, however, you want to learn how the simplex method works and see what happens in different situations using different strategies, but don't want to deal with tedious arithmetic, this module is for you!

Historically it was created to complement the Math 373 course on Mathematical Programming and Optimization at the University of Alberta, Edmonton, Canada.

AUTHORS:

- Andrey Novoseltsev (2013-03-16): initial version.
- Matthias Koeppe, Peijun Xiao (2015-07-05): allow different output styles.

EXAMPLES:

Most of the module functionality is demonstrated on the following problem.

Corn & Barley

A farmer has 1000 acres available to grow corn and barley. Corn has a net profit of 10 dollars per acre while barley has a net profit of 5 dollars per acre. The farmer has 1500 kg of fertilizer available with 3 kg per acre needed for corn and 1 kg per acre needed for barley. The farmer wants to maximize profit. (Sometimes we also add one more constraint to make the initial dictionary infeasible: the farmer has to use at least 40% of the available land.)

Using variables C and B for land used to grow corn and barley respectively, in acres, we can construct the following LP problem:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P
LP problem (use typeset mode to see details)
```

It is recommended to copy-paste such examples into your own worksheet, so that you can run these commands with typeset mode on and get

$$\begin{array}{l} \max \quad 10C + 5B \\ \qquad\quad C + B \leq 1000 \\ \qquad\quad 3C + B \leq 1500 \\ C, B \geq 0 \end{array}$$

Since it has only two variables, we can solve it graphically:

```
sage: P.plot()
Graphics object consisting of 19 graphics primitives
```

The simplex method can be applied only to `problems in standard form`, which can be created either directly

```
sage: InteractiveLPProblemStandardForm(A, b, c, ["C", "B"])
LP problem (use typeset mode to see details)
```

or from an already constructed problem of "general type":

```
sage: P = P.standard_form()
```

In this case the problem does not require any modifications to be written in standard form, but this step is still necessary to enable methods related to the simplex method.

The simplest way to use the simplex method is:

```
sage: P.run_simplex_method()
\begin{equation*}
...
The optimal value: $6250$. An optimal solution: $\left(250,\,750\right)$.
```

(This method produces quite long formulas which have been omitted here.) But, of course, it is much more fun to do most of the steps by hand. Let's start by creating the initial dictionary:

```
sage: D = P.initial_dictionary()
sage: D
LP problem dictionary (use typeset mode to see details)
```

Using typeset mode as recommended, you'll see

$$
\renewcommand{\arraystretch}{1.5}
\begin{array}{|rcrcrcr|}
\hline
x_3 & = & 1000 & - & C & - & B\\
x_4 & = & 1500 & - & 3C & - & B\\
\hline
z & = & 0 & + & 10C & + & 5B\\
\hline
\end{array}
$$

With the initial or any other dictionary you can perform a number of checks:

```
sage: D.is_feasible()
True
sage: D.is_optimal()
False
```

You can look at many of its pieces and associated data:

```
sage: D.basic_variables()
(x3, x4)
sage: D.basic_solution()
(0, 0)
sage: D.objective_value()
0
```

Most importantly, you can perform steps of the simplex method by picking an entering variable, a leaving variable, and updating the dictionary:

```
sage: D.enter("C")
sage: D.leave(4)
sage: D.update()
```

If everything was done correctly, the new dictionary is still feasible and the objective value did not decrease:

```
sage: D.is_feasible()
True
sage: D.objective_value()
5000
```

If you are unsure about picking entering and leaving variables, you can use helper methods that will try their best to tell you what are your next options:

```
sage: D.possible_entering()
[B]
sage: D.possible_leaving()
Traceback (most recent call last):
...
ValueError: leaving variables can be determined
 for feasible dictionaries with a set entering variable
 or for dual feasible dictionaries
```

It is also possible to obtain `feasible sets` and `final dictionaries` of problems, work with `revised dictionaries`, and use the dual simplex method!

Note: Currently this does not have a display format for the terminal.

9.1 Classes and functions

class `sage.numerical.interactive_simplex_method.`**`InteractiveLPProblem`**(*A*, *b*, *c*, *x='x'*, *constraint_type='<='*, *variable_type=''*, *problem_type='max'*, *base_ring=None*, *is_primal=True*)

Bases: `sage.structure.sage_object.SageObject`

Construct an LP (Linear Programming) problem.

Note: This class is for **educational purposes only**: if you want to solve Linear Programs efficiently, use `MixedIntegerLinearProgram` instead.

This class supports LP problems with "variables on the left" constraints.

INPUT:

- `A` – a matrix of constraint coefficients
- `b` – a vector of constraint constant terms
- `c` – a vector of objective coefficients

- `x` – (default: `"x"`) a vector of decision variables or a string giving the base name
- `constraint_type` – (default: `"<="`) a string specifying constraint type(s): either `"<="`, `">="`, `"=="`, or a list of them
- `variable_type` – (default: `""`) a string specifying variable type(s): either `">="`, `"<="`, `""` (the empty string), or a list of them, corresponding, respectively, to non-negative, non-positive, and free variables
- `problem_type` – (default: `"max"`) a string specifying the problem type: `"max"`, `"min"`, `"-max"`, or `"-min"`
- `base_ring` – (default: the fraction field of a common ring for all input coefficients) a field to which all input coefficients will be converted
- `is_primal` – (default: `True`) whether this problem is primal or dual: each problem is of course dual to its own dual, this flag is mostly for internal use and affects default variable names only

EXAMPLES:

We will construct the following problem:

$$\begin{array}{rrcrcl} \max & 10C & + & 5B & & \\ & C & + & B & \leq & 1000 \\ & 3C & + & B & \leq & 1500 \end{array}$$
$$C, B \geq 0$$

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
```

Same problem, but more explicitly:

```
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"],
....:     constraint_type="<=", variable_type=">=")
```

Even more explicitly:

```
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], problem_type="max",
....:     constraint_type=["<=", "<="], variable_type=[">=", ">="])
```

Using the last form you should be able to represent any LP problem, as long as all like terms are collected and in constraints variables and constants are on different sides.

A()

Return coefficients of constraints of `self`, i.e. A.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.constraint_coefficients()
[1 1]
[3 1]
sage: P.A()
[1 1]
[3 1]
```

Abcx()

Return A, b, c, and x of `self` as a tuple.

OUTPUT:

- a tuple

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.Abcx()
(
[1 1]
[3 1], (1000, 1500), (10, 5), (C, B)
)
```

b()

Return constant terms of constraints of `self`, i.e. b.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.constant_terms()
(1000, 1500)
sage: P.b()
(1000, 1500)
```

base_ring()

Return the base ring of `self`.

Note: The base ring of LP problems is always a field.

OUTPUT:

- a ring

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.base_ring()
Rational Field

sage: c = (10, 5.)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.base_ring()
Real Field with 53 bits of precision
```

c()

Return coefficients of the objective of `self`, i.e. c.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.objective_coefficients()
(10, 5)
sage: P.c()
(10, 5)
```

constant_terms()

Return constant terms of constraints of `self`, i.e. b.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.constant_terms()
(1000, 1500)
sage: P.b()
(1000, 1500)
```

constraint_coefficients()

Return coefficients of constraints of `self`, i.e. A.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.constraint_coefficients()
[1 1]
[3 1]
sage: P.A()
[1 1]
[3 1]
```

decision_variables()

Return decision variables of `self`, i.e. x.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
```

```
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.decision_variables()
(C, B)
sage: P.x()
(C, B)
```

dual(*y=None*)

Construct the dual LP problem for `self`.

INPUT:

- `y` – (default: depends on `style()`) a vector of dual decision variables or a string giving the base name

OUTPUT:

- an `InteractiveLPProblem`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: DP = P.dual()
sage: DP.b() == P.c()
True
sage: DP.dual(["C", "B"]) == P
True
```

TESTS:

```
sage: DP.standard_form().objective_name()
-z
sage: sage.numerical.interactive_simplex_method.style("Vanderbei")
'Vanderbei'
sage: P.dual().standard_form().objective_name()
-xi
sage: sage.numerical.interactive_simplex_method.style("UAlberta")
'UAlberta'
sage: P.dual().standard_form().objective_name()
-z
```

feasible_set()

Return the feasible set of `self`.

OUTPUT:

- a `Polyhedron`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.feasible_set()
A 2-dimensional polyhedron in QQ^2
defined as the convex hull of 4 vertices
```

is_bounded()

Check if `self` is bounded.

OUTPUT:

- `True` is `self` is bounded, `False` otherwise

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.is_bounded()
True
```

is_feasible()

Check if `self` is feasible.

OUTPUT:

- `True` is `self` is feasible, `False` otherwise

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.is_feasible()
True
```

is_primal()

Check if we consider this problem to be primal or dual.

This distinction affects only some automatically chosen variable names.

OUTPUT:

- boolean

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.is_primal()
True
sage: P.dual().is_primal()
False
```

m()

Return the number of constraints of `self`, i.e. m.

OUTPUT:

- an integer

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
```

```
sage: P.n_constraints()
2
sage: P.m()
2
```

n()

Return the number of decision variables of `self`, i.e. n.

OUTPUT:

- an integer

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.n_variables()
2
sage: P.n()
2
```

n_constraints()

Return the number of constraints of `self`, i.e. m.

OUTPUT:

- an integer

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.n_constraints()
2
sage: P.m()
2
```

n_variables()

Return the number of decision variables of `self`, i.e. n.

OUTPUT:

- an integer

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.n_variables()
2
sage: P.n()
2
```

objective_coefficients()

Return coefficients of the objective of `self`, i.e. c.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.objective_coefficients()
(10, 5)
sage: P.c()
(10, 5)
```

optimal_solution()

Return **an** optimal solution of `self`.

OUTPUT:

- a vector or `None` if there are no optimal solutions

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.optimal_solution()
(250, 750)
```

optimal_value()

Return the optimal value for `self`.

OUTPUT:

- a number if the problem is bounded, $\pm\infty$ if it is unbounded, or `None` if it is infeasible

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.optimal_value()
6250
```

plot(**args*, ***kwds*)

Return a plot for solving `self` graphically.

INPUT:

- `xmin`, `xmax`, `ymin`, `ymax` – bounds for the axes, if not given, an attempt will be made to pick reasonable values
- `alpha` – (default: 0.2) determines how opaque are shadows

OUTPUT:

- a plot

This only works for problems with two decision variables. On the plot the black arrow indicates the direction of growth of the objective. The lines perpendicular to it are level curves of the objective. If there are optimal solutions, the arrow originates in one of them and the corresponding level curve is solid: all points of the feasible set on it are optimal solutions. Otherwise the arrow is placed in the center. If the problem is infeasible or the objective is zero, a plot of the feasible set only is returned.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: p = P.plot()
sage: p.show()
```

In this case the plot works better with the following axes ranges:

```
sage: p = P.plot(0, 1000, 0, 1500)
sage: p.show()
```

TESTS:

We check that zero objective can be dealt with:

```
sage: InteractiveLPProblem(A, b, (0, 0), ["C", "B"], variable_type=">=").plot()
Graphics object consisting of 8 graphics primitives
```

plot_feasible_set (*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *alpha=0.2*)

Return a plot of the feasible set of `self`.

INPUT:

- `xmin`, `xmax`, `ymin`, `ymax` – bounds for the axes, if not given, an attempt will be made to pick reasonable values
- `alpha` – (default: 0.2) determines how opaque are shadows

OUTPUT:

- a plot

This only works for a problem with two decision variables. The plot shows boundaries of constraints with a shadow on one side for inequalities. If the `feasible_set()` is not empty and at least part of it is in the given boundaries, it will be shaded gray and F will be placed in its middle.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: p = P.plot_feasible_set()
sage: p.show()
```

In this case the plot works better with the following axes ranges:

```
sage: p = P.plot_feasible_set(0, 1000, 0, 1500)
sage: p.show()
```

standard_form (*objective_name=None*)

Construct the LP problem in standard form equivalent to `self`.

INPUT:

- `objective_name` – a string or a symbolic expression for the objective used in dictionaries, default depends on `style()`

OUTPUT:

- an `InteractiveLPProblemStandardForm`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: DP = P.dual()
sage: DPSF = DP.standard_form()
sage: DPSF.b()
(-10, -5)
```

x()

Return decision variables of `self`, i.e. x.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
sage: P.decision_variables()
(C, B)
sage: P.x()
(C, B)
```

class `sage.numerical.interactive_simplex_method.`**`InteractiveLPProblemStandardForm`**(*A, b, c, x='x', problem_type='max', slack_variables=None, auxiliary_variable=None, base_ring=None, is_primal=True, objective_name=None*)

Bases: `sage.numerical.interactive_simplex_method.InteractiveLPProblem`

Construct an LP (Linear Programming) problem in standard form.

Note: This class is for **educational purposes only**: if you want to solve Linear Programs efficiently, use `MixedIntegerLinearProgram` instead.

The used standard form is:

$$\pm \max cx$$
$$Ax \leq b$$
$$x \geq 0$$

INPUT:

- `A` – a matrix of constraint coefficients

- `b` – a vector of constraint constant terms
- `c` – a vector of objective coefficients
- `x` – (default: `"x"`) a vector of decision variables or a string the base name giving
- `problem_type` – (default: `"max"`) a string specifying the problem type: either `"max"` or `"-max"`
- `slack_variables` – (default: depends on `style()`) a vector of slack variables or a sting giving the base name
- `auxiliary_variable` – (default: same as `x` parameter with adjoined `"0"` if it was given as a string, otherwise `"x0"`) the auxiliary name, expected to be the same as the first decision variable for auxiliary problems
- `base_ring` – (default: the fraction field of a common ring for all input coefficients) a field to which all input coefficients will be converted
- `is_primal` – (default: `True`) whether this problem is primal or dual: each problem is of course dual to its own dual, this flag is mostly for internal use and affects default variable names only
- `objective_name` – a string or a symbolic expression for the objective used in dictionaries, default depends on `style()`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
```

Unlike `InteractiveLPProblem`, this class does not allow you to adjust types of constraints (they are always `"<="`) and variables (they are always `">="`), and the problem type may only be `"max"` or `"-max"`. You may give custom names to slack and auxiliary variables, but in most cases defaults should work:

```
sage: P.decision_variables()
(x1, x2)
sage: P.slack_variables()
(x3, x4)
```

auxiliary_problem(*objective_name=None*)

Construct the auxiliary problem for `self`.

INPUT:

- `objective_name` – a string or a symbolic expression for the objective used in dictionaries, default depends on `style()`

OUTPUT:

- an `LP problem in standard form`

The auxiliary problem with the auxiliary variable x_0 is

$$\begin{array}{l} \max -x_0 \\ -x_0 + A_i x \leq b_i \text{ for all } i \\ x \geq 0 \end{array} .$$

Such problems are used when the `initial_dictionary()` is infeasible.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
```

```
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: AP = P.auxiliary_problem()
```

auxiliary_variable()

Return the auxiliary variable of `self`.

Note that the auxiliary variable may or may not be among `decision_variables()`.

OUTPUT:

- a variable of the `coordinate_ring()` of `self`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.auxiliary_variable()
x0
sage: P.decision_variables()
(x1, x2)
sage: AP = P.auxiliary_problem()
sage: AP.auxiliary_variable()
x0
sage: AP.decision_variables()
(x0, x1, x2)
```

coordinate_ring()

Return the coordinate ring of `self`.

OUTPUT:

- a polynomial ring over the `base_ring()` of `self` in the `auxiliary_variable()`, `decision_variables()`, and `slack_variables()` with "neglex" order

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3, x4, x5
over Rational Field
sage: P.base_ring()
Rational Field
sage: P.auxiliary_variable()
x0
sage: P.decision_variables()
(x1, x2)
sage: P.slack_variables()
(x3, x4, x5)
```

dictionary(**x_B*)

Construct a dictionary for `self` with given basic variables.

INPUT:

- basic variables for the dictionary to be constructed

OUTPUT:

•a `dictionary`

Note: This is a synonym for `self.revised_dictionary(x_B).dictionary()`, but basic variables are mandatory.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary("x1", "x2")
sage: D.basic_variables()
(x1, x2)
```

feasible_dictionary(*auxiliary_dictionary*)

Construct a feasible dictionary for `self`.

INPUT:

•`auxiliary_dictionary` – an optimal dictionary for the `auxiliary_problem()` of `self` with the optimal value 0 and a non-basic auxiliary variable

OUTPUT:

•a feasible `dictionary` for `self`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: AP = P.auxiliary_problem()
sage: D = AP.initial_dictionary()
sage: D.enter(0)
sage: D.leave(5)
sage: D.update()
sage: D.enter(1)
sage: D.leave(0)
sage: D.update()
sage: D.is_optimal()
True
sage: D.objective_value()
0
sage: D.basic_solution()
(0, 400, 0)
sage: D = P.feasible_dictionary(D)
sage: D.is_optimal()
False
sage: D.is_feasible()
True
sage: D.objective_value()
4000
sage: D.basic_solution()
(400, 0)
```

final_dictionary()

Return the final dictionary of the simplex method applied to `self`.

See `run_simplex_method()` for the description of possibilities.

OUTPUT:

- a dictionary

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.final_dictionary()
sage: D.is_optimal()
True
```

TESTS:

```
sage: P.final_dictionary() is P.final_dictionary()
False
```

final_revised_dictionary()

Return the final dictionary of the revised simplex method applied to `self`.

See `run_revised_simplex_method()` for the description of possibilities.

OUTPUT:

- a `revised dictionary`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.final_revised_dictionary()
sage: D.is_optimal()
True
```

TESTS:

```
sage: P.final_revised_dictionary() is P.final_revised_dictionary()
False
```

initial_dictionary()

Construct the initial dictionary of `self`.

The initial dictionary "defines" `slack_variables()` in terms of the `decision_variables()`, i.e. it has slack variables as basic ones.

OUTPUT:

- a `dictionary`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
```

inject_variables(*scope=None*, *verbose=True*)

Inject variables of `self` into `scope`.

INPUT:

- `scope` – namespace (default: global)
- `verbose` – if `True` (default), names of injected variables will be printed

OUTPUT:

- none

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.inject_variables()
Defining x0, x1, x2, x3, x4
sage: 3*x1 + x2
x2 + 3*x1
```

objective_name()

Return the objective name used in dictionaries for this problem.

OUTPUT:

- a symbolic expression

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.objective_name()
z
sage: sage.numerical.interactive_simplex_method.style("Vanderbei")
'Vanderbei'
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.objective_name()
zeta
sage: sage.numerical.interactive_simplex_method.style("UAlberta")
'UAlberta'
sage: P = InteractiveLPProblemStandardForm(A, b, c, objective_name="custom")
sage: P.objective_name()
custom
```

revised_dictionary(**x_B*)

Construct a revised dictionary for `self`.

INPUT:

- basic variables for the dictionary to be constructed; if not given, `slack_variables()` will be used, perhaps with the `auxiliary_variable()` to give a feasible dictionary

OUTPUT:

- a `revised dictionary`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary("x1", "x2")
```

```
sage: D.basic_variables()
(x1, x2)
```

If basic variables are not given the initial dictionary is constructed:

```
sage: P.revised_dictionary().basic_variables()
(x3, x4)
sage: P.initial_dictionary().basic_variables()
(x3, x4)
```

Unless it is infeasible, in which case a feasible dictionary for the auxiliary problem is constructed:

```
sage: A = ([1, 1], [3, 1], [-1,-1])
sage: b = (1000, 1500, -400)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.initial_dictionary().is_feasible()
False
sage: P.revised_dictionary().basic_variables()
(x3, x4, x0)
```

run_revised_simplex_method()

Apply the revised simplex method and return all steps.

OUTPUT:

- `HtmlFragment` with HTML/LaTeX code of all encountered dictionaries

Note: You can access the `final_revised_dictionary()`, which can be one of the following:

- an optimal dictionary with the `auxiliary_variable()` among `basic_variables()` and a non-zero optimal value indicating that `self` is infeasible;
- a non-optimal dictionary that has marked entering variable for which there is no choice of the leaving variable, indicating that `self` is unbounded;
- an optimal dictionary.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.run_revised_simplex_method()
\begin{equation*}
...
\end{equation*}
Entering: $x_{1}$. Leaving: $x_{0}$.
\begin{equation*}
...
\end{equation*}
Entering: $x_{5}$. Leaving: $x_{4}$.
\begin{equation*}
...
\end{equation*}
Entering: $x_{2}$. Leaving: $x_{3}$.
\begin{equation*}
...
\end{equation*}
The optimal value: $6250$. An optimal solution: $\left(250,\,750\right)$.
```

run_simplex_method()

Apply the simplex method and return all steps and intermediate states.

OUTPUT:

- `HtmlFragment` with HTML/LaTeX code of all encountered dictionaries

Note: You can access the `final_dictionary()`, which can be one of the following:

- an optimal dictionary for the `auxiliary_problem()` with a non-zero optimal value indicating that `self` is infeasible;
- a non-optimal dictionary for `self` that has marked entering variable for which there is no choice of the leaving variable, indicating that `self` is unbounded;
- an optimal dictionary for `self`.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.run_simplex_method()
\begin{equation*}
...
\end{equation*}
The initial dictionary is infeasible, solving auxiliary problem.
...
Entering: $x_{0}$. Leaving: $x_{5}$.
...
Entering: $x_{1}$. Leaving: $x_{0}$.
...
Back to the original problem.
...
Entering: $x_{5}$. Leaving: $x_{4}$.
...
Entering: $x_{2}$. Leaving: $x_{3}$.
...
The optimal value: $6250$. An optimal solution: $\left(250,\,750\right)$.
```

slack_variables()

Return slack variables of `self`.

Slack variables are differences between the constant terms and left hand sides of the constraints.

If you want to give custom names to slack variables, you have to do so during construction of the problem.

OUTPUT:

- a tuple

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: P.slack_variables()
(x3, x4)
sage: P = InteractiveLPProblemStandardForm(A, b, c, ["C", "B"],
....:     slack_variables=["L", "F"])
```

```
sage: P.slack_variables()
(L, F)
```

class sage.numerical.interactive_simplex_method.**LPAbstractDictionary**

Bases: sage.structure.sage_object.SageObject

Abstract base class for dictionaries for LP problems.

Instantiating this class directly is meaningless, see LPDictionary and LPRevisedDictionary for useful extensions.

base_ring()

Return the base ring of self, i.e. the ring of coefficients.

OUTPUT:

- a ring

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.base_ring()
Rational Field
sage: D = P.revised_dictionary()
sage: D.base_ring()
Rational Field
```

basic_solution(*include_slack_variables=False*)

Return the basic solution of self.

The basic solution associated to a dictionary is obtained by setting to zero all nonbasic_variables(), in which case basic_variables() have to be equal to constant_terms() in equations. It may refer to values of decision_variables() only or include slack_variables() as well.

INPUT:

- include_slack_variables – (default: False) if True, values of slack variables will be appended at the end

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.basic_solution()
(0, 0)
sage: D.basic_solution(True)
(0, 0, 1000, 1500)
sage: D = P.revised_dictionary()
sage: D.basic_solution()
(0, 0)
```

```
sage: D.basic_solution(True)
(0, 0, 1000, 1500)
```

coordinate_ring()

Return the coordinate ring of `self`.

OUTPUT:

- a polynomial ring in `auxiliary_variable()`, `decision_variables()`, and `slack_variables()` of `self` over the `base_ring()`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3, x4
over Rational Field
sage: D = P.revised_dictionary()
sage: D.coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3, x4
over Rational Field
```

dual_ratios()

Return ratios used to determine the entering variable based on leaving.

OUTPUT:

- A list of pairs (r_j, x_j) where x_j is a non-basic variable and $r_j = c_j / a_{ij}$ is the ratio of the objective coefficient c_j to the coefficient a_{ij} of x_j in the relation for the leaving variable x_i:

$$x_i = b_i - \cdots - a_{ij} x_j - \cdots .$$

The order of pairs matches the order of `nonbasic_variables()`, but only x_j with negative a_{ij} are considered.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary(2, 3, 5)
sage: D.leave(3)
sage: D.dual_ratios()
[(5/2, x1), (5, x4)]
sage: D = P.revised_dictionary(2, 3, 5)
sage: D.leave(3)
sage: D.dual_ratios()
[(5/2, x1), (5, x4)]
```

enter(*v*)

Set `v` as the entering variable of `self`.

INPUT:

- `v` – a non-basic variable of `self`, can be given as a string, an actual variable, or an integer interpreted as the index of a variable. It is also possible to enter `None` to reset choice.

OUTPUT:

- none, but the selected variable will be used as entering by methods that require an entering variable and the corresponding column will be typeset in green

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.enter("x1")
```

We can also use indices of variables:

```
sage: D.enter(1)
```

Or variable names without quotes after injecting them:

```
sage: P.inject_variables()
Defining x0, x1, x2, x3, x4
sage: D.enter(x1)
```

The same works for revised dictionaries as well:

```
sage: D = P.revised_dictionary()
sage: D.enter(x1)
```

entering()

Return the currently chosen entering variable.

OUTPUT:

- a variable if the entering one was chosen, otherwise `None`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.entering() is None
True
sage: D.enter(1)
sage: D.entering()
x1
```

is_dual_feasible()

Check if `self` is dual feasible.

OUTPUT:

- `True` if all `objective_coefficients()` are non-positive, `False` otherwise

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.is_dual_feasible()
```

```
False
sage: D = P.revised_dictionary()
sage: D.is_dual_feasible()
False
```

is_feasible()

Check if `self` is feasible.

OUTPUT:

- `True` if all `constant_terms()` are non-negative, `False` otherwise

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.is_feasible()
True
sage: D = P.revised_dictionary()
sage: D.is_feasible()
True
```

is_optimal()

Check if `self` is optimal.

OUTPUT:

- `True` if `self` `is_feasible()` and `is_dual_feasible()` (i.e. all `constant_terms()` are non-negative and all `objective_coefficients()` are non-positive), `False` otherwise.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.is_optimal()
False
sage: D = P.revised_dictionary()
sage: D.is_optimal()
False
sage: D = P.revised_dictionary(1, 2)
sage: D.is_optimal()
True
```

leave(*v*)

Set `v` as the leaving variable of `self`.

INPUT:

- `v` – a basic variable of `self`, can be given as a string, an actual variable, or an integer interpreted as the index of a variable. It is also possible to leave `None` to reset choice.

OUTPUT:

- none, but the selected variable will be used as leaving by methods that require a leaving variable and the corresponding row will be typeset in red

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.leave("x4")
```

We can also use indices of variables:

```
sage: D.leave(4)
```

Or variable names without quotes after injecting them:

```
sage: P.inject_variables()
Defining x0, x1, x2, x3, x4
sage: D.leave(x4)
```

The same works for revised dictionaries as well:

```
sage: D = P.revised_dictionary()
sage: D.leave(x4)
```

leaving()

Return the currently chosen leaving variable.

OUTPUT:

- a variable if the leaving one was chosen, otherwise `None`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.leaving() is None
True
sage: D.leave(4)
sage: D.leaving()
x4
```

possible_dual_simplex_method_steps()

Return possible dual simplex method steps for `self`.

OUTPUT:

- A list of pairs `(leaving, entering)`, where `leaving` is a basic variable that may `leave()` and `entering` is a list of non-basic variables that may `enter()` when `leaving` leaves. Note that `entering` may be empty, indicating that the problem is infeasible (since the dual one is unbounded).

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary(2, 3)
sage: D.possible_dual_simplex_method_steps()
[(x3, [x1])]
sage: D = P.revised_dictionary(2, 3)
```

```
sage: D.possible_dual_simplex_method_steps()
[(x3, [x1])]
```

possible_entering()

Return possible entering variables for `self`.

OUTPUT:

- a list of non-basic variables of `self` that can `enter()` on the next step of the (dual) simplex method

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.possible_entering()
[x1, x2]
sage: D = P.revised_dictionary()
sage: D.possible_entering()
[x1, x2]
```

possible_leaving()

Return possible leaving variables for `self`.

OUTPUT:

- a list of basic variables of `self` that can `leave()` on the next step of the (dual) simplex method

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.enter(1)
sage: D.possible_leaving()
[x4]
sage: D = P.revised_dictionary()
sage: D.enter(1)
sage: D.possible_leaving()
[x4]
```

possible_simplex_method_steps()

Return possible simplex method steps for `self`.

OUTPUT:

- A list of pairs `(entering, leaving)`, where `entering` is a non-basic variable that may `enter()` and `leaving` is a list of basic variables that may `leave()` when `entering` enters. Note that `leaving` may be empty, indicating that the problem is unbounded.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.possible_simplex_method_steps()
```

```
[(x1, [x4]), (x2, [x3])]
sage: D = P.revised_dictionary()
sage: D.possible_simplex_method_steps()
[(x1, [x4]), (x2, [x3])]
```

ratios()

Return ratios used to determine the leaving variable based on entering.

OUTPUT:

- A list of pairs (r_i, x_i) where x_i is a basic variable and $r_i = b_i / a_{ik}$ is the ratio of the constant term b_i to the coefficient a_{ik} of the entering variable x_k in the relation for x_i:

$$x_i = b_i - \cdots - a_{ik} x_k - \cdots .$$

The order of pairs matches the order of `basic_variables()`, but only x_i with positive a_{ik} are considered.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.enter(1)
sage: D.ratios()
[(1000, x3), (500, x4)]
sage: D = P.revised_dictionary()
sage: D.enter(1)
sage: D.ratios()
[(1000, x3), (500, x4)]
```

run_dual_simplex_method()

Apply the dual simplex method and return all steps/intermediate states.

If either entering or leaving variables were already set, they will be used.

OUTPUT:

- `HtmlFragment` with HTML/LaTeX code of all encountered dictionaries

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.run_dual_simplex_method()
Traceback (most recent call last):
...
ValueError: leaving variables can be determined for feasible
dictionaries with a set entering variable or for dual feasible
dictionaries
```

Let's start with a dual feasible dictionary then:

```
sage: D = P.dictionary(2, 3, 5)
sage: D.is_dual_feasible()
True
sage: D.is_optimal()
```

```
False
sage: D.run_dual_simplex_method()
\begin{equation*}
...
\end{equation*}
Leaving: $x_{3}$. Entering: $x_{1}$.
\begin{equation*}
...
\end{equation*}
sage: D.is_optimal()
True
```

This method detects infeasible problems:

```
sage: A = ([1, 0],)
sage: b = (-1,)
sage: c = (0, -1)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.run_dual_simplex_method()
\begin{equation*}
...
\end{equation*}
The problem is infeasible because of $x_{3}$ constraint.
```

run_simplex_method()

Apply the simplex method and return all steps and intermediate states.

If either entering or leaving variables were already set, they will be used.

OUTPUT:

- `HtmlFragment` with HTML/LaTeX code of all encountered dictionaries

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.run_simplex_method()
Traceback (most recent call last):
...
ValueError: entering variables can be determined for feasible
dictionaries or for dual feasible dictionaries with a set leaving
variable
```

Let's start with a feasible dictionary then:

```
sage: D = P.dictionary(1, 3, 4)
sage: D.is_feasible()
True
sage: D.is_optimal()
False
sage: D.run_simplex_method()
\begin{equation*}
...
\end{equation*}
Entering: $x_{5}$. Leaving: $x_{4}$.
\begin{equation*}
```

```
...
\end{equation*}
Entering: $x_{2}$. Leaving: $x_{3}$.
\begin{equation*}
...
\end{equation*}
sage: D.is_optimal()
True
```

This method detects unbounded problems:

```
sage: A = ([1, 0],)
sage: b = (1,)
sage: c = (0, 1)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.run_simplex_method()
\begin{equation*}
...
\end{equation*}
The problem is unbounded in $x_{2}$ direction.
```

class `sage.numerical.interactive_simplex_method.`**`LPDictionary`**(*A*, *b*, *c*, *objective_value*, *basic_variables*, *nonbasic_variables*, *objective_name*)

Bases: `sage.numerical.interactive_simplex_method.LPAbstractDictionary`

Construct a dictionary for an LP problem.

A dictionary consists of the following data:

$$\boxed{\begin{array}{l} x_B = b - Ax_N \\ \hline z = z^* + cx_N \end{array}}$$

INPUT:

- `A` – a matrix of relation coefficients
- `b` – a vector of relation constant terms
- `c` – a vector of objective coefficients
- `objective_value` – current value of the objective z^*
- `basic_variables` – a list of basic variables x_B
- `nonbasic_variables` – a list of non-basic variables x_N
- `objective_name` – a “name” for the objective z

OUTPUT:

- a `dictionary for an LP problem`

Note: This constructor does not check correctness of input, as it is intended to be used internally by `InteractiveLPProblemStandardForm`.

EXAMPLES:

The intended way to use this class is indirect:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D
LP problem dictionary (use typeset mode to see details)
```

But if you want you can create a dictionary without starting with an LP problem, here is construction of the same dictionary as above:

```
sage: A = matrix(QQ, ([1, 1], [3, 1]))
sage: b = vector(QQ, (1000, 1500))
sage: c = vector(QQ, (10, 5))
sage: R = PolynomialRing(QQ, "x1, x2, x3, x4", order="neglex")
sage: from sage.numerical.interactive_simplex_method \
....:     import LPDictionary
sage: D2 = LPDictionary(A, b, c, 0, R.gens()[2:], R.gens()[:2], "z")
sage: D2 == D
True
```

ELLUL (*entering*, *leaving*)

Perform the Enter-Leave-LaTeX-Update-LaTeX step sequence on `self`.

INPUT:

- `entering` – the entering variable
- `leaving` – the leaving variable

OUTPUT:

- a string with LaTeX code for `self` before and after update

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.ELLUL("x1", "x4")
doctest:...: DeprecationWarning: ELLUL is deprecated, please use separate enter-leave-update
See http://trac.sagemath.org/19097 for details.
\renewcommand{\arraystretch}{1.5} %notruncate
\begin{array}{|rcrcrcr|}
\hline
x_{3} \mspace{-6mu}&\mspace{-6mu} = \mspace{-6mu}&\mspace{-6mu} 1000 \mspace{-6mu}&\mspace{-
\color{red}x_{4} \mspace{-6mu}&\color{red}\mspace{-6mu} = \mspace{-6mu}&\color{red}\mspace{-
\hline
z \mspace{-6mu}&\mspace{-6mu} = \mspace{-6mu}&\mspace{-6mu} 0 \mspace{-6mu}&\mspace{-6mu} +
\hline
\\
\hline
x_{3} \mspace{-6mu}&\mspace{-6mu} = \mspace{-6mu}&\mspace{-6mu} 500 \mspace{-6mu}&\mspace{-6
x_{1} \mspace{-6mu}&\mspace{-6mu} = \mspace{-6mu}&\mspace{-6mu} 500 \mspace{-6mu}&\mspace{-6
\hline
z \mspace{-6mu}&\mspace{-6mu} = \mspace{-6mu}&\mspace{-6mu} 5000 \mspace{-6mu}&\mspace{-6mu}
\hline
\end{array}
```

This is how the above output looks when rendered:

$$\begin{array}{|rcrcrcr|}
\hline
x_{3} & = & 1000 & - & x_{1} & - & x_{2}\\
x_{4} & = & 1500 & - & 3 x_{1} & - & x_{2}\\
\hline
z & = & 0 & + & 10 x_{1} & + & 5 x_{2}\\
\hline
\\
\hline
x_{3} & = & 500 & + & \frac{1}{3} x_{4} & - & \frac{2}{3} x_{2}\\
x_{1} & = & 500 & - & \frac{1}{3} x_{4} & - & \frac{1}{3} x_{2}\\
\hline
z & = & 5000 & - & \frac{10}{3} x_{4} & + & \frac{5}{3} x_{2}\\
\hline
\end{array}$$

The column of the entering variable is green, while the row of the leaving variable is red in the original dictionary state on the top. The new state after the update step is shown on the bottom.

basic_variables()

Return the basic variables of `self`.

OUTPUT:

•a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.basic_variables()
(x3, x4)
```

constant_terms()

Return the constant terms of relations of `self`.

OUTPUT:

•a vector.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.constant_terms()
(1000, 1500)
```

entering_coefficients()

Return coefficients of the entering variable.

OUTPUT:

•a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
```

```
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.enter(1)
sage: D.entering_coefficients()
(1, 3)
```

leaving_coefficients()

Return coefficients of the leaving variable.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary(2, 3)
sage: D.leave(3)
sage: D.leaving_coefficients()
(-2, -1)
```

nonbasic_variables()

Return non-basic variables of `self`.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.nonbasic_variables()
(x1, x2)
```

objective_coefficients()

Return coefficients of the objective of `self`.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.objective_coefficients()
(10, 5)
```

objective_value()

Return the value of the objective at the `basic_solution()` of `self`.

OUTPUT:

•a number

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.objective_value()
0
```

update()

Update `self` using previously set entering and leaving variables.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.objective_value()
0
sage: D.enter("x1")
sage: D.leave("x4")
sage: D.update()
sage: D.objective_value()
5000
```

`sage.numerical.interactive_simplex_method.`**LPProblem**

alias of `InteractiveLPProblem`

`sage.numerical.interactive_simplex_method.`**LPProblemStandardForm**

alias of `InteractiveLPProblemStandardForm`

class `sage.numerical.interactive_simplex_method.`**LPRevisedDictionary**(*problem*, *basic_variables*)

Bases: `sage.numerical.interactive_simplex_method.LPAbstractDictionary`

Construct a revised dictionary for an LP problem.

INPUT:

•`problem` – an `LP problem in standard form`

•`basic_variables` – a list of basic variables or their indices

OUTPUT:

•a `revised dictionary for an LP problem`

A revised dictionary encodes the same relations as a `regular dictionary`, but stores only what is "necessary to efficiently compute data for the simplex method".

Let the original problem be

$$\pm \max cx$$
$$Ax \leq b$$
$$x \geq 0$$

Let $\bar{x}$ be the vector of `decision_variables()` x followed by the `slack_variables()`. Let $\bar{c}$ be the vector of `objective_coefficients()` c followed by zeroes for all slack variables. Let $\bar{A} = (A|I)$ be the matrix of `constraint_coefficients()` A augmented by the identity matrix as columns corresponding to the slack variables. Then the problem above can be written as

$$\begin{aligned}&\pm\max \bar{c}\bar{x}\\ &\bar{A}\bar{x} = b\\ &\bar{x} \geq 0\end{aligned}$$

and any dictionary is a system of equations equivalent to $\bar{A}\bar{x} = b$, but resolved for `basic_variables()` x_B in terms of `nonbasic_variables()` x_N together with the expression for the objective in terms of x_N. Let `c_B()` and `c_N()` be vectors "splitting $\bar{c}$ into basic and non-basic parts". Let `B()` and `A_N()` be the splitting of $\bar{A}$. Then the corresponding dictionary is

$$\begin{array}{|l|}\hline x_B = B^{-1}b - B^{-1}A_N x_N\\ \hline z = yb + \left(c_N - y^T A_N\right) x_N\\ \hline\end{array}$$

where $y = c_B^T B^{-1}$. To proceed with the simplex method, it is not necessary to compute all entries of this dictionary. On the other hand, any entry is easy to compute, if you know B^{-1}, so we keep track of it through the update steps.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: from sage.numerical.interactive_simplex_method \
....:     import LPRevisedDictionary
sage: D = LPRevisedDictionary(P, [1, 2])
sage: D.basic_variables()
(x1, x2)
sage: D
LP problem dictionary (use typeset mode to see details)
```

The same dictionary can be constructed through the problem:

```
sage: P.revised_dictionary(1, 2) == D
True
```

When this dictionary is typeset, you will see two tables like these ones:

$$\begin{array}{l|r|rr||r||r}
x_B & c_B & & B^{-1} & y & B^{-1}b\\ \hline
x_{1} & 10 & -\frac{1}{2} & \frac{1}{2} & \frac{5}{2} & 250\\
x_{2} & 5 & \frac{3}{2} & -\frac{1}{2} & \frac{5}{2} & 750
\end{array}$$

$$\begin{array}{r|rr}
x_N & x_{3} & x_{4}\\ \hline
c_N^T & 0 & 0\\ \hline
y^T A_N & \frac{5}{2} & \frac{5}{2}\\ \hline
c_N^T - y^T A_N & -\frac{5}{2} & -\frac{5}{2}
\end{array}$$

More details will be shown if entering and leaving variables are set, but in any case the top table shows B^{-1} and a few extra columns, while the bottom one shows several rows: these are related to columns and rows of dictionary entries.

A(*v*)

Return the column of constraint coefficients corresponding to `v`.

INPUT:

- `v` – a variable, its name, or its index

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.A(1)
(1, 3)
sage: D.A(0)
(-1, -1)
sage: D.A("x3")
(1, 0)
```

A_N()

Return the A_N matrix, constraint coefficients of non-basic variables.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.A_N()
[1 1]
[3 1]
```

B()

Return the B matrix, i.e. constraint coefficients of basic variables.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary(1, 2)
sage: D.B()
[1 1]
[3 1]
```

B_inverse()

Return the inverse of the `B()` matrix.

This inverse matrix is stored and computed during dictionary update in a more efficient way than generic inversion.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary(1, 2)
sage: D.B_inverse()
[-1/2  1/2]
[ 3/2 -1/2]
```

E()

Return the eta matrix between `self` and the next dictionary.

OUTPUT:

- a matrix

If B_{old} is the current matrix B and B_{new} is the B matrix of the next dictionary (after the update step), then $B_{\mathrm{new}} = B_{\mathrm{old}} E$.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.enter(1)
sage: D.leave(4)
sage: D.E()
[1 1]
[0 3]
```

E_inverse()

Return the inverse of the matrix `E()`.

This inverse matrix is computed in a more efficient way than generic inversion.

OUTPUT:

- a matrix

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.enter(1)
sage: D.leave(4)
sage: D.E_inverse()
[   1 -1/3]
[   0  1/3]
```

basic_indices()

Return the basic indices of `self`.

Note: Basic indices are indices of `basic_variables()` in the list of generators of the `coordinate_ring()` of the `problem()` of `self`, they may not coincide with the indices of variables which are parts of their names. (They will for the default indexed names.)

OUTPUT:

- a list.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.basic_indices()
[3, 4]
```

basic_variables()

Return the basic variables of `self`.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.basic_variables()
(x3, x4)
```

c_B()

Return the c_B vector, objective coefficients of basic variables.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary(1, 2)
sage: D.c_B()
(10, 5)
```

c_N()

Return the c_N vector, objective coefficients of non-basic variables.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
```

```
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.c_N()
(10, 5)
```

constant_terms()

Return constant terms in the relations of `self`.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.constant_terms()
(1000, 1500)
```

dictionary()

Return a regular LP dictionary matching `self`.

OUTPUT:

- an `LP dictionary`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1], [-1, -1])
sage: b = (1000, 1500, -400)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.dictionary()
LP problem dictionary (use typeset mode to see details)
```

entering_coefficients()

Return coefficients of the entering variable.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.enter(1)
sage: D.entering_coefficients()
(1, 3)
```

leaving_coefficients()

Return coefficients of the leaving variable.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary(2, 3)
sage: D.leave(3)
sage: D.leaving_coefficients()
(-2, -1)
```

nonbasic_indices()

Return the non-basic indices of `self`.

Note: Non-basic indices are indices of `nonbasic_variables()` in the list of generators of the `coordinate_ring()` of the `problem()` of `self`, they may not coincide with the indices of variables which are parts of their names. (They will for the default indexed names.)

OUTPUT:

- a list

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.nonbasic_indices()
[1, 2]
```

nonbasic_variables()

Return non-basic variables of `self`.

OUTPUT:

- a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.nonbasic_variables()
(x1, x2)
```

objective_coefficients()

Return coefficients of the objective of `self`.

OUTPUT:

- a vector

These are coefficients of non-basic variables when basic variables are eliminated.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
```

```
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.objective_coefficients()
(10, 5)
```

objective_value()

Return the value of the objective at the basic solution of `self`.

OUTPUT:

- a number

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.objective_value()
0
```

problem()

Return the original problem.

OUTPUT:

- an `LP problem in standard form`

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.problem() is P
True
```

update()

Update `self` using previously set entering and leaving variables.

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.objective_value()
0
sage: D.enter("x1")
sage: D.leave("x4")
sage: D.update()
sage: D.objective_value()
5000
```

x_B()

Return the basic variables of `self`.

OUTPUT:

• a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.basic_variables()
(x3, x4)
```

x_N()

Return non-basic variables of `self`.

OUTPUT:

• a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.nonbasic_variables()
(x1, x2)
```

y()

Return the y vector, the product of `c_B()` and `B_inverse()`.

OUTPUT:

• a vector

EXAMPLES:

```
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.revised_dictionary()
sage: D.y()
(0, 0)
```

sage.numerical.interactive_simplex_method.**default_variable_name**(*variable*)

Return default variable name for the current `style()`.

INPUT:

• `variable` - a string describing requested name

OUTPUT:

• a string with the requested name for current style

EXAMPLES:

```
sage: sage.numerical.interactive_simplex_method.default_variable_name("primal slack")
'x'
sage: sage.numerical.interactive_simplex_method.style('Vanderbei')
'Vanderbei'
sage: sage.numerical.interactive_simplex_method.default_variable_name("primal slack")
```

```
'w'
sage: sage.numerical.interactive_simplex_method.style('UAlberta')
'UAlberta'
```

sage.numerical.interactive_simplex_method.**random_dictionary**(*m*, *n*, *bound=5*, *special_probability=0.2*)

Construct a random dictionary.

INPUT:

- m – the number of constraints/basic variables
- n – the number of decision/non-basic variables
- bound – (default: 5) a bound on dictionary entries
- special_probability – (default: 0.2) probability of constructing a potentially infeasible or potentially optimal dictionary

OUTPUT:

- an LP problem dictionary

EXAMPLES:

```
sage: from sage.numerical.interactive_simplex_method \
....:     import random_dictionary
sage: random_dictionary(3, 4)
LP problem dictionary (use typeset mode to see details)
```

sage.numerical.interactive_simplex_method.**style**(*new_style=None*)

Set or get the current style of problems and dictionaries.

INPUT:

- new_style – a string or None (default)

OUTPUT:

- a string with current style (same as new_style if it was given)

If the input is not recognized as a valid style, a ValueError exception is raised.

Currently supported styles are:

- 'UAlberta' (default): Follows the style used in the Math 373 course on Mathematical Programming and Optimization at the University of Alberta, Edmonton, Canada; based on Chvatal's book.
  - Objective functions of dictionaries are printed at the bottom.

  Variable names default to

  - z for primal objective
  - z for dual objective
  - w for auxiliary objective
  - $x_1, x_2, \dots, x_n$ for primal decision variables
  - $x_{n+1}, x_{n+2}, \dots, x_{n+m}$ for primal slack variables
  - $y_1, y_2, \dots, y_m$ for dual decision variables
  - $y_{m+1}, y_{m+2}, \dots, y_{m+n}$ for dual slack variables

- 'Vanderbei': Follows the style of Robert Vanderbei's textbook, Linear Programming – Foundations and Extensions.
  - Objective functions of dictionaries are printed at the top.

  Variable names default to
  - $zeta$ for primal objective
  - xi for dual objective
  - xi for auxiliary objective
  - $x_1, x_2, \ldots, x_n$ for primal decision variables
  - $w_1, w_2, \ldots, w_m$ for primal slack variables
  - $y_1, y_2, \ldots, y_m$ for dual decision variables
  - $z_1, z_2, \ldots, z_n$ for dual slack variables

EXAMPLES:

```
sage: sage.numerical.interactive_simplex_method.style()
'UAlberta'
sage: sage.numerical.interactive_simplex_method.style('Vanderbei')
'Vanderbei'
sage: sage.numerical.interactive_simplex_method.style('Doesntexist')
Traceback (most recent call last):
...
ValueError: Style must be one of: UAlberta, Vanderbei
sage: sage.numerical.interactive_simplex_method.style('UAlberta')
'UAlberta'
```

`sage.numerical.interactive_simplex_method.`**`variable`**`(`*R*, *v*`)`

Interpret `v` as a variable of `R`.

INPUT:

- `R` – a polynomial ring
- `v` – a variable of `R` or convertible into `R`, a string with the name of a variable of `R` or an index of a variable in `R`

OUTPUT:

- a variable of `R`

EXAMPLES:

```
sage: from sage.numerical.interactive_simplex_method \
....:     import variable
sage: R = PolynomialRing(QQ, "x3, y5, x5, y")
sage: R.inject_variables()
Defining x3, y5, x5, y
sage: variable(R, "x3")
x3
sage: variable(R, x3)
x3
sage: variable(R, 3)
x3
sage: variable(R, 0)
Traceback (most recent call last):
...
ValueError: there is no variable with the given index
```

```
sage: variable(R, 5)
Traceback (most recent call last):
...
ValueError: the given index is ambiguous
sage: variable(R, 2 * x3)
Traceback (most recent call last):
...
ValueError: cannot interpret given data as a variable
sage: variable(R, "z")
Traceback (most recent call last):
...
ValueError: cannot interpret given data as a variable
```

CHAPTER

TEN

LINEAR OPTIMIZATION (LP) SOLVER BACKENDS

10.1 Generic Backend for LP solvers

This class only lists the methods that should be defined by any interface with a LP Solver. All these methods immediately raise `NotImplementedError` exceptions when called, and are obviously meant to be replaced by the solver-specific method. This file can also be used as a template to create a new interface : one would only need to replace the occurrences of `"Nonexistent_LP_solver"` by the solver's name, and replace `GenericBackend` by `SolverName(GenericBackend)` so that the new solver extends this class.

AUTHORS:

- Nathann Cohen (2010-10) : initial implementation
- Risan (2012-02) : extension for PPL backend
- Ingolfur Edvardsson (2014-06): extension for CVXOPT backend

class `sage.numerical.backends.generic_backend.`**`GenericBackend`**

Bases: `object`

`add_col`(*indices*, *coeffs*)

Add a column.

INPUT:

- `indices` (list of integers) – this list contains the indices of the constraints in which the variable's coefficient is nonzero
- `coeffs` (list of real values) – associates a coefficient to the variable in each of the constraints in which it appears. Namely, the i-th entry of `coeffs` corresponds to the coefficient of the variable in the constraint represented by the i-th entry in `indices`.

Note: `indices` and `coeffs` are expected to be of the same length.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.ncols()                                        # optional - Nonexistent_LP_solver
0
sage: p.nrows()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_linear_constraints(5, 0, None)             # optional - Nonexistent_LP_solver
sage: p.add_col(range(5), range(5))                    # optional - Nonexistent_LP_solver
sage: p.nrows()                                        # optional - Nonexistent_LP_solver
5
```

add_linear_constraint (*coefficients*, *lower_bound*, *upper_bound*, *name=None*)

Add a linear constraint.

INPUT:

- `coefficients` – an iterable of pairs `(i, v)`. In each pair, `i` is a variable index (integer) and `v` is a value (element of `base_ring()`).
- `lower_bound` – element of `base_ring()` or `None`. The lower bound.
- `upper_bound` – element of `base_ring()` or `None`. The upper bound.
- `name` – string or `None`. Optional name for this row.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import GenericBackend
sage: solver = GenericBackend()
sage: solver.add_linear_constraint(zip(range(5), range(5)), 2.0, 2.0)
Traceback (most recent call last):
...
NotImplementedError: add_linear_constraint
```

add_linear_constraint_vector (*degree*, *coefficients*, *lower_bound*, *upper_bound*, *name=None*)

Add a vector-valued linear constraint.

Note: This is the generic implementation, which will split the vector-valued constraint into components and add these individually. Backends are encouraged to replace it with their own optimized implementation.

INPUT:

- `degree` – integer. The vector degree, that is, the number of new scalar constraints.
- `coefficients` – an iterable of pairs `(i, v)`. In each pair, `i` is a variable index (integer) and `v` is a vector (real and of length `degree`).
- `lower_bound` – either a vector or `None`. The component-wise lower bound.
- `upper_bound` – either a vector or `None`. The component-wise upper bound.
- `name` – string or `None`. An optional name for all new rows.

EXAMPLE:

```
sage: coeffs = ([0, vector([1, 2])], [1, vector([2, 3])])
sage: upper = vector([5, 5])
sage: lower = vector([0, 0])
sage: from sage.numerical.backends.generic_backend import GenericBackend
sage: solver = GenericBackend()
sage: solver.add_linear_constraint_vector(2, coeffs, lower, upper, 'foo')
Traceback (most recent call last):
...
NotImplementedError: add_linear_constraint
```

add_linear_constraints (*number*, *lower_bound*, *upper_bound*, *names=None*)

Add constraints.

INPUT:

- `number` (integer) – the number of constraints to add.
- `lower_bound` - a lower bound, either a real value or `None`

- upper_bound - an upper bound, either a real value or None
- names - an optional list of names (default: None)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")    # optional - Nonexistent_LP_solver
sage: p.add_variables(5)                                  # optional - Nonexistent_LP_solver
5
sage: p.add_linear_constraints(5, None, 2)            # optional - Nonexistent_LP_solver
sage: p.row(4)                                        # optional - Nonexistent_LP_solver
([], [])
sage: p.row_bounds(4)                                 # optional - Nonexistent_LP_solver
(None, 2.0)
```

add_variable(*lower_bound=None*, *upper_bound=None*, *binary=False*, *continuous=True*, *integer=False*, *obj=None*, *name=None*)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both positive and real.

INPUT:

- lower_bound - the lower bound of the variable (default: 0)
- upper_bound - the upper bound of the variable (default: None)
- binary - True if the variable is binary (default: False).
- continuous - True if the variable is binary (default: True).
- integer - True if the variable is binary (default: False).
- obj - (optional) coefficient of this variable in the objective function (default: 0.0)
- name - an optional name for the newly added variable (default: None).

OUTPUT: The index of the newly created variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")     # optional - Nonexistent_LP_solver
sage: p.ncols()                                            # optional - Nonexistent_LP_solver
0
sage: p.add_variable()                                     # optional - Nonexistent_LP_solver
0
sage: p.ncols()                                            # optional - Nonexistent_LP_solver
1
sage: p.add_variable(binary=True)                          # optional - Nonexistent_LP_solver
1
sage: p.add_variable(lower_bound=-2.0, integer=True)       # optional - Nonexistent_LP_solver
2
sage: p.add_variable(continuous=True, integer=True)        # optional - Nonexistent_LP_solver
Traceback (most recent call last):
...
ValueError: ...
sage: p.add_variable(name='x',obj=1.0)                     # optional - Nonexistent_LP_solver
3
sage: p.col_name(3)                                        # optional - Nonexistent_LP_solver
'x'
sage: p.objective_coefficient(3)                           # optional - Nonexistent_LP_solver
1.0
```

add_variables (*n*, *lower_bound=None*, *upper_bound=None*, *binary=False*, *continuous=True*, *integer=False*, *obj=None*, *names=None*)

Add n variables.

This amounts to adding new columns to the matrix. By default, the variables are both positive and real.

INPUT:

- n - the number of new variables (must be > 0)
- lower_bound - the lower bound of the variable (default: 0)
- upper_bound - the upper bound of the variable (default: None)
- binary - True if the variable is binary (default: False).
- continuous - True if the variable is binary (default: True).
- integer - True if the variable is binary (default: False).
- obj - (optional) coefficient of all variables in the objective function (default: 0.0)
- names - optional list of names (default: None)

OUTPUT: The index of the variable created last.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")     # optional - Nonexistent_LP_solver
sage: p.ncols()                                            # optional - Nonexistent_LP_solver
0
sage: p.add_variables(5)                                   # optional - Nonexistent_LP_solver
4
sage: p.ncols()                                            # optional - Nonexistent_LP_solver
5
sage: p.add_variables(2, lower_bound=-2.0, integer=True, names=['a','b']) # optional - Nonex
6
```

base_ring ()

best_known_objective_bound ()

Return the value of the currently best known bound.

This method returns the current best upper (resp. lower) bound on the optimal value of the objective function in a maximization (resp. minimization) problem. It is equal to the output of get_objective_value() if the MILP found an optimal solution, but it can differ if it was interrupted manually or after a time limit (cf solver_parameter()).

Note: Has no meaning unless solve has been called before.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver") # optional - Nonexistent
sage: b = p.new_variable(binary=True)                       # optional - Nonexistent_LP_solve
sage: for u,v in graphs.CycleGraph(5).edges(labels=False): # optional - Nonexistent_LP_solve
....:     p.add_constraint(b[u]+b[v]<=1)                   # optional - Nonexistent_LP_solve
sage: p.set_objective(p.sum(b[x] for x in range(5)))       # optional - Nonexistent_LP_solve
sage: p.solve()                                             # optional - Nonexistent_LP_solve
2.0
sage: pb = p.get_backend()                                  # optional - Nonexistent_LP_solve
sage: pb.get_objective_value()                              # optional - Nonexistent_LP_solve
2.0
```

```
sage: pb.best_known_objective_bound()                    # optional - Nonexistent_LP_solve
2.0
```

col_bounds (*index*)

Return the bounds of a specific variable.

INPUT:

- `index` (integer) – the variable's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the variable is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variable()                                 # optional - Nonexistent_LP_solver
1
sage: p.col_bounds(0)                              # optional - Nonexistent_LP_solver
(0.0, None)
sage: p.variable_upper_bound(0, 5)                 # optional - Nonexistent_LP_solver
sage: p.col_bounds(0)                              # optional - Nonexistent_LP_solver
(0.0, 5.0)
```

col_name (*index*)

Return the `index`-th column name

INPUT:

- `index` (integer) – the column id
- `name` (`char *`) – its name. When set to `NULL` (default), the method returns the current name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variable(name="I am a variable")            # optional - Nonexistent_LP_solver
1
sage: p.col_name(0)                                     # optional - Nonexistent_LP_solver
'I am a variable'
```

get_objective_value ()

Return the value of the objective function.

Note: Behavior is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver") # optional - Nonexistent_LP_solver
sage: p.add_variables(2)                               # optional - Nonexistent_LP_solver
2
sage: p.add_linear_constraint([(0,1), (1,2)], None, 3) # optional - Nonexistent_LP_solver
sage: p.set_objective([2, 5])                          # optional - Nonexistent_LP_solver
sage: p.solve()                                        # optional - Nonexistent_LP_solver
0
sage: p.get_objective_value()                          # optional - Nonexistent_LP_solver
7.5
```

```
sage: p.get_variable_value(0)                             # optional - Nonexistent_LP_solver
0.0
sage: p.get_variable_value(1)                             # optional - Nonexistent_LP_solver
1.5
```

get_relative_objective_gap()

Return the relative objective gap of the best known solution.

For a minimization problem, this value is computed by $(\texttt{bestinteger} - \texttt{bestobjective})/(1e-10 + |\texttt{bestobjective}|)$, where `bestinteger` is the value returned by `get_objective_value()` and `bestobjective` is the value returned by `best_known_objective_bound()`. For a maximization problem, the value is computed by $(\texttt{bestobjective} - \texttt{bestinteger})/(1e - 10 + |\texttt{bestobjective}|)$.

Note: Has no meaning unless `solve` has been called before.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver") # optional - Nonexistent
sage: b = p.new_variable(binary=True)                     # optional - Nonexistent_LP_solve
sage: for u,v in graphs.CycleGraph(5).edges(labels=False): # optional - Nonexistent_LP_solve
....:     p.add_constraint(b[u]+b[v]<=1)                  # optional - Nonexistent_LP_solve
sage: p.set_objective(p.sum(b[x] for x in range(5)))      # optional - Nonexistent_LP_solve
sage: p.solve()                                           # optional - Nonexistent_LP_solve
2.0
sage: pb = p.get_backend()                                # optional - Nonexistent_LP_solve
sage: pb.get_objective_value()                            # optional - Nonexistent_LP_solve
2.0
sage: pb.get_best_objective_value()                       # optional - Nonexistent_LP_solve
2.0
sage: pb.get_relative_objective_gap()                     # optional - Nonexistent_LP_solve
0.0
```

get_variable_value(*variable*)

Return the value of a variable given by the solver.

Note: Behavior is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver") # optional - Nonexistent_LP_solver
sage: p.add_variables(2)                              # optional - Nonexistent_LP_solver
2
sage: p.add_linear_constraint([(0,1), (1, 2)], None, 3) # optional - Nonexistent_LP_solver
sage: p.set_objective([2, 5])                         # optional - Nonexistent_LP_solver
sage: p.solve()                                       # optional - Nonexistent_LP_solver
0
sage: p.get_objective_value()                         # optional - Nonexistent_LP_solver
7.5
sage: p.get_variable_value(0)                         # optional - Nonexistent_LP_solver
0.0
sage: p.get_variable_value(1)                         # optional - Nonexistent_LP_solver
1.5
```

is_maximization()

Test whether the problem is a maximization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver") # optional - Nonexistent_LP_solver
sage: p.is_maximization()                              # optional - Nonexistent_LP_solver
True
sage: p.set_sense(-1)                              # optional - Nonexistent_LP_solver
sage: p.is_maximization()                              # optional - Nonexistent_LP_solver
False
```

is_variable_binary(*index*)

Test whether the given variable is of binary type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.ncols()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
1
sage: p.set_variable_type(0,0)                          # optional - Nonexistent_LP_solver
sage: p.is_variable_binary(0)                           # optional - Nonexistent_LP_solver
True
```

is_variable_continuous(*index*)

Test whether the given variable is of continuous/real type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.ncols()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
1
sage: p.is_variable_continuous(0)                       # optional - Nonexistent_LP_solver
True
sage: p.set_variable_type(0,1)                          # optional - Nonexistent_LP_solver
sage: p.is_variable_continuous(0)                       # optional - Nonexistent_LP_solver
False
```

is_variable_integer(*index*)

Test whether the given variable is of integer type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.ncols()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
```

```
1
sage: p.set_variable_type(0,1)                        # optional - Nonexistent_LP_solver
sage: p.is_variable_integer(0)                        # optional - Nonexistent_LP_solver
True
```

ncols()

Return the number of columns/variables.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.ncols()                                       # optional - Nonexistent_LP_solver
0
sage: p.add_variables(2)                              # optional - Nonexistent_LP_solver
2
sage: p.ncols()                                       # optional - Nonexistent_LP_solver
2
```

nrows()

Return the number of rows/constraints.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver") # optional - Nonexistent_LP_solver
sage: p.nrows()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_linear_constraints(2, 2.0, None)         # optional - Nonexistent_LP_solver
sage: p.nrows()                                      # optional - Nonexistent_LP_solver
2
```

objective_coefficient(*variable*, *coeff=None*)

Set or get the coefficient of a variable in the objective function

INPUT:

- `variable` (integer) – the variable's id
- `coeff` (double) – its coefficient

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variable()                                 # optional - Nonexistent_LP_solver
1
sage: p.objective_coefficient(0)                         # optional - Nonexistent_LP_solver
0.0
sage: p.objective_coefficient(0,2)                       # optional - Nonexistent_LP_solver
sage: p.objective_coefficient(0)                         # optional - Nonexistent_LP_solver
2.0
```

problem_name(*name='NULL'*)

Return or define the problem's name

INPUT:

- `name (char *)` – the problem's name. When set to `NULL` (default), the method returns the problem's name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")   # optional - Nonexistent_LP_solver
sage: p.problem_name("There once was a french fry") # optional - Nonexistent_LP_solver
sage: print p.get_problem_name()                        # optional - Nonexistent_LP_solver
There once was a french fry
```

remove_constraint (*i*)

Remove a constraint.

INPUT:

- i – index of the constraint to remove.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_constraint(p[0] + p[1], max = 10)           # optional - Nonexistent_LP_solver
sage: p.remove_constraint(0)                            # optional - Nonexistent_LP_solver
```

remove_constraints (*constraints*)

Remove several constraints.

INPUT:

- constraints – an iterable containing the indices of the rows to remove.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_constraint(p[0] + p[1], max = 10)           # optional - Nonexistent_LP_solver
sage: p.remove_constraints([0])                         # optional - Nonexistent_LP_solver
```

row (*i*)

Return a row

INPUT:

- index (integer) – the constraint's id.

OUTPUT:

A pair (indices, coeffs) where indices lists the entries whose coefficient is nonzero, and to which coeffs associates their coefficient on the model of the add_linear_constraint method.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variables(5)                               # optional - Nonexistent_LP_solver
5
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2) # optional - Nonexistent_LP_sol
sage: p.row(0)                                     # optional - Nonexistent_LP_solver
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
sage: p.row_bounds(0)                              # optional - Nonexistent_LP_solver
(2.0, 2.0)
```

row_bounds (*index*)

Return the bounds of a specific constraint.

INPUT:

• `index` (integer) – the constraint's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variables(5)                               # optional - Nonexistent_LP_solver
5
sage: p.add_linear_constraint(range(5), range(5), 2, 2) # optional - Nonexistent_LP_solver
sage: p.row(0)                                     # optional - Nonexistent_LP_solver
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
sage: p.row_bounds(0)                              # optional - Nonexistent_LP_solver
(2.0, 2.0)
```

row_name (*index*)

Return the `index` th row name

INPUT:

• `index` (integer) – the row's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_linear_constraints(1, 2, None, name="Empty constraint 1")  # optional - Nonexist
sage: p.row_name(0)                                     # optional - Nonexistent_LP_solver
'Empty constraint 1'
```

set_objective (*coeff*, *d=0.0*)

Set the objective function.

INPUT:

• `coeff` – a list of real values, whose i-th element is the coefficient of the i-th variable in the objective function.

• `d` (double) – the constant term in the linear function (set to 0 by default)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")    # optional - Nonexistent_LP_solver
sage: p.add_variables(5)                                 # optional - Nonexistent_LP_solver
5
sage: p.set_objective([1, 1, 2, 1, 3])                   # optional - Nonexistent_LP_solver
sage: map(lambda x :p.objective_coefficient(x), range(5))  # optional - Nonexistent_LP_solve
[1.0, 1.0, 2.0, 1.0, 3.0]
```

Constants in the objective function are respected:

```
sage: p = MixedIntegerLinearProgram(solver='Nonexistent_LP_solver') # optional - Nonexistent
sage: x,y = p[0], p[1]                              # optional - Nonexistent_LP_solver
sage: p.add_constraint(2*x + 3*y, max = 6)          # optional - Nonexistent_LP_solver
sage: p.add_constraint(3*x + 2*y, max = 6)          # optional - Nonexistent_LP_solver
sage: p.set_objective(x + y + 7)                    # optional - Nonexistent_LP_solver
sage: p.set_integer(x); p.set_integer(y)            # optional - Nonexistent_LP_solver
sage: p.solve()                                     # optional - Nonexistent_LP_solver
9.0
```

set_sense (*sense*)
Set the direction (maximization/minimization).

INPUT:

- `sense` (integer) :
  - +1 => Maximization
  - −1 => Minimization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.is_maximization()                              # optional - Nonexistent_LP_solver
True
sage: p.set_sense(-1)                              # optional - Nonexistent_LP_solver
sage: p.is_maximization()                              # optional - Nonexistent_LP_solver
False
```

set_variable_type (*variable*, *vtype*)
Set the type of a variable

INPUT:

- `variable` (integer) – the variable's id
- `vtype` (integer) :
  - 1 Integer
  - 0 Binary
  - **-1** Continuous

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")   # optional - Nonexistent_LP_solver
sage: p.ncols()                                        # optional - Nonexistent_LP_solver
0
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
1
sage: p.set_variable_type(0,1)                          # optional - Nonexistent_LP_solver
sage: p.is_variable_integer(0)                          # optional - Nonexistent_LP_solver
True
```

set_verbosity (*level*)
Set the log (verbosity) level

INPUT:

- `level` (integer) – From 0 (no verbosity) to 3.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.set_verbosity(2)                                # optional - Nonexistent_LP_solver
```

solve ()
Solve the problem.

Note: This method raises `MIPSolverException` exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver") # optional - Nonexistent_LP_solver
sage: p.add_linear_constraints(5, 0, None)              # optional - Nonexistent_LP_solver
sage: p.add_col(range(5), range(5))                     # optional - Nonexistent_LP_solver
sage: p.solve()                                         # optional - Nonexistent_LP_solver
0
sage: p.objective_coefficient(0,1)                    # optional - Nonexistent_LP_solver
sage: p.solve()                                         # optional - Nonexistent_LP_solver
Traceback (most recent call last):
...
MIPSolverException: ...
```

solver_parameter(*name*, *value=None*)

Return or define a solver parameter

INPUT:

- `name` (string) – the parameter
- `value` – the parameter's value if it is to be defined, or `None` (default) to obtain its current value.

Note: The list of available parameters is available at `solver_parameter()`.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.solver_parameter("timelimit")                    # optional - Nonexistent_LP_solver
sage: p.solver_parameter("timelimit", 60)                # optional - Nonexistent_LP_solver
sage: p.solver_parameter("timelimit")                    # optional - Nonexistent_LP_solver
```

variable_lower_bound(*index*, *value=None*)

Return or define the lower bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not lower bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
1
sage: p.col_bounds(0)                                 # optional - Nonexistent_LP_solver
(0.0, None)
sage: p.variable_lower_bound(0, 5)                    # optional - Nonexistent_LP_solver
sage: p.col_bounds(0)                                 # optional - Nonexistent_LP_solver
(5.0, None)
```

variable_upper_bound(*index*, *value=None*)

Return or define the upper bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not upper bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variable()                                  # optional - Nonexistent_LP_solver
1
sage: p.col_bounds(0)                              # optional - Nonexistent_LP_solver
(0.0, None)
sage: p.variable_upper_bound(0, 5)                 # optional - Nonexistent_LP_solver
sage: p.col_bounds(0)                              # optional - Nonexistent_LP_solver
(0.0, 5.0)
```

write_lp(*name*)

Write the problem to a `.lp` file

INPUT:

- `filename` (string)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variables(2)                                # optional - Nonexistent_LP_solver
2
sage: p.add_linear_constraint([(0, 1], (1, 2)], None, 3) # optional - Nonexistent_LP_solver
sage: p.set_objective([2, 5])                           # optional - Nonexistent_LP_solver
sage: p.write_lp(os.path.join(SAGE_TMP, "lp_problem.lp"))            # optional - Nonexisten
```

write_mps(*name*, *modern*)

Write the problem to a `.mps` file

INPUT:

- `filename` (string)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
sage: p.add_variables(2)                                # optional - Nonexistent_LP_solver
2
sage: p.add_linear_constraint([(0, 1), (1, 2)], None, 3) # optional - Nonexistent_LP_solver
sage: p.set_objective([2, 5])                           # optional - Nonexistent_LP_solver
sage: p.write_lp(os.path.join(SAGE_TMP, "lp_problem.lp"))            # optional - Nonexisten
```

zero()

`sage.numerical.backends.generic_backend.`**default_mip_solver**(*solver=None*)

Returns/Sets the default MILP Solver used by Sage

INPUT:

- `solver` – defines the solver to use:
  - GLPK (`solver="GLPK"`). See the GLPK web site.

- COIN Branch and Cut (`solver="Coin"`). See the COIN-OR web site.
- CPLEX (`solver="CPLEX"`). See the CPLEX web site.
- CVXOPT (`solver="CVXOPT"`). See the CVXOPT web site.
- PPL (`solver="PPL"`). See the PPL web site.
- Gurobi (`solver="Gurobi"`). See the Gurobi web site.

`solver` should then be equal to one of `"GLPK"`, `"Coin"`, `"CPLEX"`, `"CVXOPT"`, `"Gurobi"` or `"PPL"` .

- If `solver=None` (default), the current default solver's name is returned.

OUTPUT:

This function returns the current default solver's name if `solver = None` (default). Otherwise, it sets the default solver to the one given. If this solver does not exist, or is not available, a `ValueError` exception is raised.

EXAMPLE:

```
sage: former_solver = default_mip_solver()
sage: default_mip_solver("GLPK")
sage: default_mip_solver()
'Glpk'
sage: default_mip_solver("PPL")
sage: default_mip_solver()
'Ppl'
sage: default_mip_solver("GUROBI")
Traceback (most recent call last):
...
ValueError: Gurobi is not available. Please refer to the documentation to install it.
sage: default_mip_solver("Yeahhhhhhhhhhh")
Traceback (most recent call last):
...
ValueError: 'solver' should be set to 'GLPK', 'Coin', 'CPLEX', 'Gurobi', 'CVXOPT', 'PPL' or None
sage: default_mip_solver(former_solver)
```

`sage.numerical.backends.generic_backend.`**`get_solver`**(*constraint_generation=False*, *solver=None*)

Return a solver according to the given preferences

INPUT:

- `solver` – 6 solvers should be available through this class:
  - GLPK (`solver="GLPK"`). See the GLPK web site.
  - COIN Branch and Cut (`solver="Coin"`). See the COIN-OR web site.
  - CPLEX (`solver="CPLEX"`). See the CPLEX web site.
  - CVXOPT (`solver="CVXOPT"`). See the CVXOPT web site.
  - Gurobi (`solver="Gurobi"`). See the Gurobi web site.
  - PPL (`solver="PPL"`). See the PPL web site.

  `solver` should then be equal to one of `"GLPK"`, `"Coin"`, `"CPLEX"`, `"CVXOPT"`,‘“Gurobi”, ``"PPL"`, or `None`. If `solver=None` (default), the default solver is used (see `default_mip_solver` method.
- `constraint_generation` – Only used when `solver=None`.

–When set to True, after solving the MixedIntegerLinearProgram, it is possible to add a constraint, and then solve it again. The effect is that solvers that do not support this feature will not be used.

–Defaults to False.

See also:

•default_mip_solver() – Returns/Sets the default MIP solver.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver()
```

10.2 GLPK Backend

AUTHORS:

- Nathann Cohen (2010-10): initial implementation
- John Perry (2012-01): glp_simplex preprocessing
- John Perry and Raniere Gaia Silva (2012-03): solver parameters
- Christian Kuper (2012-10): Additions for sensitivity analysis

class sage.numerical.backends.glpk_backend.**GLPKBackend**

Bases: sage.numerical.backends.generic_backend.GenericBackend

add_col (*indices*, *coeffs*)

Add a column.

INPUT:

•indices (list of integers) – this list constains the indices of the constraints in which the variable's coefficient is nonzero

•coeffs (list of real values) – associates a coefficient to the variable in each of the constraints in which it appears. Namely, the ith entry of coeffs corresponds to the coefficient of the variable in the constraint represented by the ith entry in indices.

Note: indices and coeffs are expected to be of the same length.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.nrows()
0
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(range(5), range(5))
sage: p.nrows()
5
```

add_linear_constraint (*coefficients*, *lower_bound*, *upper_bound*, *name=None*)

Add a linear constraint.

INPUT:

- `coefficients` an iterable with `(c,v)` pairs where `c` is a variable index (integer) and `v` is a value (real value).
- `lower_bound` - a lower bound, either a real value or `None`
- `upper_bound` - an upper bound, either a real value or `None`
- `name` - an optional name for this row (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint( zip(range(5), range(5)), 2.0, 2.0)
sage: p.row(0)
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
sage: p.row_bounds(0)
(2.0, 2.0)
sage: p.add_linear_constraint( zip(range(5), range(5)), 1.0, 1.0, name='foo')
sage: p.row_name(1)
'foo'
```

add_linear_constraints(*number*, *lower_bound*, *upper_bound*, *names=None*)

Add '`number` linear constraints.

INPUT:

- `number` (integer) – the number of constraints to add.
- `lower_bound` - a lower bound, either a real value or `None`
- `upper_bound` - an upper bound, either a real value or `None`
- `names` - an optional list of names (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(5, None, 2)
sage: p.row(4)
([], [])
sage: p.row_bounds(4)
(None, 2.0)
sage: p.add_linear_constraints(2, None, 2, names=['foo','bar'])
```

add_variable(*lower_bound=0.0*, *upper_bound=None*, *binary=False*, *continuous=False*, *integer=False*, *obj=0.0*, *name=None*)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both positive, real and the coefficient in the objective function is 0.0.

INPUT:

- `lower_bound` - the lower bound of the variable (default: 0)
- `upper_bound` - the upper bound of the variable (default: `None`)

- binary - True if the variable is binary (default: False).
- continuous - True if the variable is binary (default: True).
- integer - True if the variable is binary (default: False).
- obj - (optional) coefficient of this variable in the objective function (default: 0.0)
- name - an optional name for the newly added variable (default: None).

OUTPUT: The index of the newly created variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.ncols()
1
sage: p.add_variable(binary=True)
1
sage: p.add_variable(lower_bound=-2.0, integer=True)
2
sage: p.add_variable(continuous=True, integer=True)
Traceback (most recent call last):
...
ValueError: ...
sage: p.add_variable(name='x',obj=1.0)
3
sage: p.col_name(3)
'x'
sage: p.objective_coefficient(3)
1.0
```

add_variables (*number*, *lower_bound=0.0*, *upper_bound=None*, *binary=False*, *continuous=False*, *integer=False*, *obj=0.0*, *names=None*)

Add number new variables.

This amounts to adding new columns to the matrix. By default, the variables are both positive, real and theor coefficient in the objective function is 0.0.

INPUT:

- n - the number of new variables (must be > 0)
- lower_bound - the lower bound of the variable (default: 0)
- upper_bound - the upper bound of the variable (default: None)
- binary - True if the variable is binary (default: False).
- continuous - True if the variable is binary (default: True).
- integer - True if the variable is binary (default: False).
- obj - (optional) coefficient of all variables in the objective function (default: 0.0)
- names - optional list of names (default: None)

OUTPUT: The index of the variable created last.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variables(5)
4
sage: p.ncols()
5
sage: p.add_variables(2, lower_bound=-2.0, integer=True, names=['a','b'])
6
```

best_known_objective_bound()

Return the value of the currently best known bound.

This method returns the current best upper (resp. lower) bound on the optimal value of the objective function in a maximization (resp. minimization) problem. It is equal to the output of `get_objective_value()` if the MILP found an optimal solution, but it can differ if it was interrupted manually or after a time limit (cf `solver_parameter()`).

Note: Has no meaning unless `solve` has been called before.

EXAMPLE:

```
sage: g = graphs.CubeGraph(9)
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: p.solver_parameter("mip_gap_tolerance",100)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(p.sum(b[v] for v in g))
sage: for v in g:
....:     p.add_constraint(b[v]+p.sum(b[u] for u in g.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1) # Force an easy non-0 solution
sage: p.solve() # rel tol 100
1.0
sage: backend = p.get_backend()
sage: backend.best_known_objective_bound() # random
48.0
```

col_bounds(*index*)

Return the bounds of a specific variable.

INPUT:

- `index` (integer) – the variable's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the variable is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0.0, 5.0)
```

col_name(*index*)

Return the `index` th col name

INPUT:

- `index` (integer) – the col's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variable(name='I am a variable')
0
sage: p.col_name(0)
'I am a variable'
```

copy()

Returns a copy of self.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: b = p.new_variable()
sage: p.add_constraint(b[1] + b[2] <= 6)
sage: p.set_objective(b[1] + b[2])
sage: copy(p).solve()
6.0
```

eval_tab_col(*k*)

Computes a column of the current simplex tableau.

A (column) corresponds to some non-basic variable specified by the parameter `k` as follows:

- if $0 \leq k \leq m - 1$, the non-basic variable is k-th auxiliary variable,
- if $m \leq k \leq m + n - 1$, the non-basic variable is $(k - m)$-th structual variable,

where m is the number of rows and n is the number of columns in the specified problem object.

Note: The basis factorization must exist. Otherwise a `MIPSolverException` will be raised.

INPUT:

- `k` (integer) – the id of the non-basic variable.

OUTPUT:

A pair `(indices, coeffs)` where `indices` lists the entries whose coefficient is nonzero, and to which `coeffs` associates their coefficient in the computed column of the current simplex tableau.

Note: Elements in `indices` have the same sense as index k. All these variables are basic by definition.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
```

```
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.eval_tab_col(1)
Traceback (most recent call last):
...
MIPSolverException: ...
sage: lp.solve()
0
sage: lp.eval_tab_col(1)
([0, 5, 3], [-2.0, 2.0, -0.5])
sage: lp.eval_tab_col(2)
([0, 5, 3], [8.0, -4.0, 1.5])
sage: lp.eval_tab_col(4)
([0, 5, 3], [-2.0, 2.0, -1.25])
sage: lp.eval_tab_col(0)
Traceback (most recent call last):
...
MIPSolverException: ...
sage: lp.eval_tab_col(-1)
Traceback (most recent call last):
...
ValueError: ...
```

eval_tab_row(*k*)

Computes a row of the current simplex tableau.

A row corresponds to some basic variable specified by the parameter `k` as follows:

- if $0 \leq k \leq m-1$, the basic variable is k-th auxiliary variable,
- if $m \leq k \leq m+n-1$, the basic variable is $(k-m)$-th structual variable,

where m is the number of rows and n is the number of columns in the specified problem object.

Note: The basis factorization must exist. Otherwise, a `MIPSolverException` will be raised.

INPUT:

- `k` (integer) – the id of the basic variable.

OUTPUT:

A pair `(indices, coeffs)` where `indices` lists the entries whose coefficient is nonzero, and to which `coeffs` associates their coefficient in the computed row of the current simplex tableau.

Note: Elements in `indices` have the same sense as index `k`. All these variables are non-basic by definition.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
```

```
sage: lp.eval_tab_row(0)
Traceback (most recent call last):
...
MIPSolverException: ...
sage: lp.solve()
0
sage: lp.eval_tab_row(0)
([1, 2, 4], [-2.0, 8.0, -2.0])
sage: lp.eval_tab_row(3)
([1, 2, 4], [-0.5, 1.5, -1.25])
sage: lp.eval_tab_row(5)
([1, 2, 4], [2.0, -4.0, 2.0])
sage: lp.eval_tab_row(1)
Traceback (most recent call last):
...
MIPSolverException: ...
sage: lp.eval_tab_row(-1)
Traceback (most recent call last):
...
ValueError: ...
```

get_col_dual(*variable*)

Returns the dual value (reduced cost) of a variable

The dual value is the reduced cost of a variable. The reduced cost is the amount by which the objective coefficient of a non basic variable has to change to become a basic variable.

INPUT:

- `variable` – The number of the variable

Note: Behaviour is undefined unless `solve` has been called before. If the simplex algorithm has not been used for solving just a 0.0 will be returned.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(3)
2
sage: p.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: p.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: p.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: p.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: p.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: p.solve()
0
sage: p.get_col_dual(1)
-5.0
```

get_col_stat(*j*)

Retrieve the status of a variable.

INPUT:

- `j` – The index of the variable

OUTPUT:

- Returns current status assigned to the structural variable associated with j-th column:
  - GLP_BS = 1 basic variable
  - GLP_NL = 2 non-basic variable on lower bound
  - GLP_NU = 3 non-basic variable on upper bound
  - GLP_NF = 4 non-basic free (unbounded) variable
  - GLP_NS = 5 non-basic fixed variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_col_stat(0)
1
sage: lp.get_col_stat(1)
2
sage: lp.get_col_stat(-1)
Exception ValueError: ...
0
```

get_objective_value()

Returns the value of the objective function.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: p.get_objective_value()
7.5
sage: p.get_variable_value(0) # abs tol 1e-15
0.0
sage: p.get_variable_value(1)
1.5
```

get_relative_objective_gap()

Return the relative objective gap of the best known solution.

For a minimization problem, this value is computed by $(\texttt{bestinteger} - \texttt{bestobjective})/(1e-10 + |\texttt{bestobjective}|)$, where `bestinteger` is the value returned by `get_objective_value()`

and `bestobjective` is the value returned by `best_known_objective_bound()`. For a maximization problem, the value is computed by $(\texttt{bestobjective} - \texttt{bestinteger})/(1e - 10 + |\texttt{bestobjective}|)$.

Note: Has no meaning unless `solve` has been called before.

EXAMPLE:

```
sage: g = graphs.CubeGraph(9)
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: p.solver_parameter("mip_gap_tolerance",100)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(p.sum(b[v] for v in g))
sage: for v in g:
....:     p.add_constraint(b[v]+p.sum(b[u] for u in g.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1) # Force an easy non-0 solution
sage: p.solve() # rel tol 100
1.0
sage: backend = p.get_backend()
sage: backend.get_relative_objective_gap() # random
46.99999999999999
```

TESTS:

Just make sure that the variable *has* been defined, and is not just undefined:

```
sage: backend.get_relative_objective_gap() > 1
True
```

get_row_dual(*variable*)

Returns the dual value of a constraint.

The dual value of the ith row is also the value of the ith variable of the dual problem.

The dual value of a constraint is the shadow price of the constraint. The shadow price is the amount by which the objective value will change if the constraints bounds change by one unit under the precondition that the basis remains the same.

INPUT:

- `variable` – The number of the constraint

Note: Behaviour is undefined unless `solve` has been called before. If the simplex algorithm has not been used for solving 0.0 will be returned.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_row_dual(0)    # tolerance 0.00001
```

```
0.0
sage: lp.get_row_dual(1)    # tolerance 0.00001
10.0
```

get_row_prim(*i*)

Returns the value of the auxiliary variable associated with i-th row.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_objective_value()
280.0
sage: lp.get_row_prim(0)
24.0
sage: lp.get_row_prim(1)
20.0
sage: lp.get_row_prim(2)
8.0
```

get_row_stat(*i*)

Retrieve the status of a constraint.

INPUT:

- `i` – The index of the constraint

OUTPUT:

- Returns current status assigned to the auxiliary variable associated with i-th row:
  - GLP_BS = 1 basic variable
  - GLP_NL = 2 non-basic variable on lower bound
  - GLP_NU = 3 non-basic variable on upper bound
  - GLP_NF = 4 non-basic free (unbounded) variable
  - GLP_NS = 5 non-basic fixed variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
```

```
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_row_stat(0)
1
sage: lp.get_row_stat(1)
3
sage: lp.get_row_stat(-1)
Exception ValueError: ...
0
```

get_variable_value(*variable*)

Returns the value of a variable given by the solver.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: p.get_objective_value()
7.5
sage: p.get_variable_value(0) # abs tol 1e-15
0.0
sage: p.get_variable_value(1)
1.5
```

is_maximization()

Test whether the problem is a maximization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

is_variable_binary(*index*)

Test whether the given variable is of binary type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
```

```
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,0)
sage: p.is_variable_binary(0)
True
```

is_variable_continuous(*index*)

Test whether the given variable is of continuous/real type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_continuous(0)
True
sage: p.set_variable_type(0,1)
sage: p.is_variable_continuous(0)
False
```

is_variable_integer(*index*)

Test whether the given variable is of integer type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,1)
sage: p.is_variable_integer(0)
True
```

ncols()

Return the number of columns/variables.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()
Return the number of rows/constraints.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.nrows()
0
sage: p.add_linear_constraints(2, 2, None)
sage: p.nrows()
2
```

objective_coefficient(*variable*, *coeff=None*)
Set or get the coefficient of a variable in the objective function

INPUT:

- `variable` (integer) – the variable's id
- `coeff` (double) – its coefficient or `None` for reading (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0.0
sage: p.objective_coefficient(0,2)
sage: p.objective_coefficient(0)
2.0
```

print_ranges(*filename='NULL'*)
Print results of a sensitivity analysis

If no filename is given as an input the results of the sensitivity analysis are displayed on the screen. If a filename is given they are written to a file.

INPUT:

- `filename` – (optional) name of the file

OUTPUT:

Zero if the operations was successful otherwise nonzero.

Note: This method is only effective if an optimal solution has been found for the lp using the simplex algorithm. In all other cases an error message is printed.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint(zip([0, 1], [1, 2]), None, 3)
sage: p.set_objective([2, 5])
sage: import sage.numerical.backends.glpk_backend as backend
sage: p.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: p.print_ranges()
glp_print_ranges: optimal basic solution required
```

```
1
sage: p.solve()
0
sage: p.print_ranges()
Write sensitivity analysis report to...
GLPK ... - SENSITIVITY ANALYSIS REPORT

Problem:
Objective:  7.5 (MAXimum)

   No. Row name     St      Activity         Slack   Lower bound       Activity      Obj coe
                                          Marginal   Upper bound          range         rang
------ ------------ -- ------------- ------------- -------------  ------------- ------------
     1              NU       3.00000             .          -Inf              .     -2.5000
                                           2.50000       3.00000           +Inf         +In

GLPK ... - SENSITIVITY ANALYSIS REPORT

Problem:
Objective:  7.5 (MAXimum)

   No. Column name  St      Activity      Obj coef   Lower bound       Activity      Obj coe
                                          Marginal   Upper bound          range         rang
------ ------------ -- ------------- ------------- -------------  ------------- ------------
     1              NL             .       2.00000             .           -Inf          -In
                                          -.50000          +Inf        3.00000       2.5000

     2              BS       1.50000       5.00000             .           -Inf       4.0000
                                                 .          +Inf        1.50000          +In

End of report

0
```

problem_name (*name='NULL'*)

Return or define the problem's name

INPUT:

- `name` (`char *`) – the problem's name. When set to `NULL` (default), the method returns the problem's name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.problem_name("There once was a french fry")
sage: print p.problem_name()
There once was a french fry
```

remove_constraint (*i*)

Remove a constraint from self.

INPUT:

- `i` – index of the constraint to remove

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x, y = p['x'], p['y']
```

```
sage: p.add_constraint(2*x + 3*y <= 6)
sage: p.add_constraint(3*x + 2*y <= 6)
sage: p.add_constraint(x >= 0)
sage: p.set_objective(x + y + 7)
sage: p.set_integer(x); p.set_integer(y)
sage: p.solve()
9.0
sage: p.remove_constraint(0)
sage: p.solve()
10.0
```

Removing fancy constraints does not make Sage crash:

```
sage: MixedIntegerLinearProgram(solver = "GLPK").remove_constraint(-2)
Traceback (most recent call last):
...
ValueError: The constraint's index i must satisfy 0 <= i < number_of_constraints
```

remove_constraints(*constraints*)

Remove several constraints.

INPUT:

- `constraints` – an iterable containing the indices of the rows to remove.

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x, y = p['x'], p['y']
sage: p.add_constraint(2*x + 3*y <= 6)
sage: p.add_constraint(3*x + 2*y <= 6)
sage: p.add_constraint(x >= 0)
sage: p.set_objective(x + y + 7)
sage: p.set_integer(x); p.set_integer(y)
sage: p.solve()
9.0
sage: p.remove_constraints([0])
sage: p.solve()
10.0
sage: p.get_values([x,y])
[0.0, 3.0]
```

TESTS:

Removing fancy constraints does not make Sage crash:

```
sage: MixedIntegerLinearProgram(solver= "GLPK").remove_constraints([0, -2])
Traceback (most recent call last):
...
ValueError: The constraint's index i must satisfy 0 <= i < number_of_constraints
```

row(*index*)

Return a row

INPUT:

- `index` (integer) – the constraint's id.

OUTPUT:

A pair `(indices, coeffs)` where `indices` lists the entries whose coefficient is nonzero, and to which `coeffs` associates their coefficient on the model of the `add_linear_constraint` method.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
sage: p.row_bounds(0)
(2.0, 2.0)
```

row_bounds (*index*)

Return the bounds of a specific constraint.

INPUT:

- `index` (integer) – the constraint's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
sage: p.row_bounds(0)
(2.0, 2.0)
```

row_name (*index*)

Return the `index` th row name

INPUT:

- `index` (integer) – the row's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_linear_constraints(1, 2, None, names=['Empty constraint 1'])
sage: p.row_name(0)
'Empty constraint 1'
```

set_col_stat (*j, stat*)

Set the status of a variable.

INPUT:

- `j` – The index of the constraint
- `stat` – The status to set to

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
```

```
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_col_stat(0)
1
sage: lp.set_col_stat(0, 2)
sage: lp.get_col_stat(0)
2
```

set_objective (*coeff, d=0.0*)

Set the objective function.

INPUT:

- `coeff` - a list of real values, whose ith element is the coefficient of the ith variable in the objective function.
- `d` (double) – the constant term in the linear function (set to 0 by default)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
sage: map(lambda x :p.objective_coefficient(x), range(5))
[1.0, 1.0, 2.0, 1.0, 3.0]
```

set_row_stat (*i, stat*)

Set the status of a constraint.

INPUT:

- `i` – The index of the constraint
- `stat` – The status to set to

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_row_stat(0)
1
sage: lp.set_row_stat(0, 3)
```

```
sage: lp.get_row_stat(0)
3
```

set_sense(*sense*)

Set the direction (maximization/minimization).

INPUT:

- `sense` (integer) :
  - +1 => Maximization
  - −1 => Minimization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

set_variable_type(*variable*, *vtype*)

Set the type of a variable

INPUT:

- `variable` (integer) – the variable's id
- `vtype` (integer) :
  - 1 Integer
  - 0 Binary
  - −1 Real

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,1)
sage: p.is_variable_integer(0)
True
```

set_verbosity(*level*)

Set the verbosity level

INPUT:

- `level` (integer) – From 0 (no verbosity) to 3.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.set_verbosity(2)
```

solve()

Solve the problem.

Sage uses GLPK's implementation of the branch-and-cut algorithm (`glp_intopt`) to solve the mixed-integer linear program. This algorithm can be requested explicitly by setting the solver parameter "simplex_or_intopt" to "intopt_only". (If all variables are continuous, the algorithm reduces to solving the linear program by the simplex method.)

EXAMPLE:

```
sage: lp = MixedIntegerLinearProgram(solver = 'GLPK', maximization = False)
sage: x, y = lp[0], lp[1]
sage: lp.add_constraint(-2*x + y <= 1)
sage: lp.add_constraint(x - y <= 1)
sage: lp.add_constraint(x + y >= 2)
sage: lp.set_objective(x + y)
sage: lp.set_integer(x)
sage: lp.set_integer(y)
sage: lp.solve()
2.0
sage: lp.get_values([x, y])
[1.0, 1.0]
```

Note: This method raises `MIPSolverException` exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(range(5), range(5))
sage: p.solve()
0
sage: p.objective_coefficient(0,1)
sage: p.solve()
Traceback (most recent call last):
...
MIPSolverException: ...
```

Warning: Sage uses GLPK's `glp_intopt` to find solutions. This routine sometimes FAILS CATASTROPHICALLY when given a system it cannot solve. (Ticket #12309.) Here, "catastrophic" can mean either "infinite loop" or segmentation fault. Upstream considers this behavior "essentially innate" to their design, and suggests preprocessing it with `glp_simplex` first. Thus, if you suspect that your system is infeasible, set the `preprocessing` option first.

EXAMPLE:

```
sage: lp = MixedIntegerLinearProgram(solver = "GLPK")
sage: v = lp.new_variable(nonnegative=True)
sage: lp.add_constraint(v[1] +v[2] -2.0 *v[3], max=-1.0)
sage: lp.add_constraint(v[0] -4.0/3 *v[1] +1.0/3 *v[2], max=-1.0/3)
sage: lp.add_constraint(v[0] +0.5 *v[1] -0.5 *v[2] +0.25 *v[3], max=-0.25)
sage: lp.solve()
0.0
sage: lp.add_constraint(v[0] +4.0 *v[1] -v[2] +v[3], max=-1.0)
sage: lp.solve()
Traceback (most recent call last):
```

```
...
RuntimeError: GLPK : Signal sent, try preprocessing option
sage: lp.solver_parameter("simplex_or_intopt", "simplex_then_intopt")
sage: lp.solve()
Traceback (most recent call last):
...
MIPSolverException: 'GLPK : Problem has no feasible solution'
```

If we switch to "simplex_only", the integrality constraints are ignored, and we get an optimal solution to the continuous relaxation.

EXAMPLE:

```
sage: lp = MixedIntegerLinearProgram(solver = 'GLPK', maximization = False)
sage: x, y = lp[0], lp[1]
sage: lp.add_constraint(-2*x + y <= 1)
sage: lp.add_constraint(x - y <= 1)
sage: lp.add_constraint(x + y >= 2)
sage: lp.set_objective(x + y)
sage: lp.set_integer(x)
sage: lp.set_integer(y)
sage: lp.solver_parameter("simplex_or_intopt", "simplex_only") # use simplex only
sage: lp.solve()
2.0
sage: lp.get_values([x, y])
[1.5, 0.5]
```

If one solves a linear program and wishes to access dual information ($get_{c}ol_{d}ual$ etc.) or tableau data ($get_{r}ow_{s}tat$ etc.), one needs to switch to "simplex_only" before solving.

GLPK also has an exact rational simplex solver. The only access to data is via double-precision floats, however. It reconstructs rationals from doubles and also provides results as doubles.

EXAMPLE:

```
sage: lp.solver_parameter("simplex_or_intopt", "exact_simplex_only") # use exact simplex onl
sage: lp.solve()
glp_exact: 3 rows, 2 columns, 6 non-zeros
GNU MP bignum library is being used
...
OPTIMAL SOLUTION FOUND
2.0
sage: lp.get_values([x, y])
[1.5, 0.5]
```

If you need the rational solution, you need to retrieve the basis information via `get_col_stat` and `get_row_stat` and calculate the corresponding basic solution. Below we only test that the basis information is indeed available. Calculating the corresponding basic solution is left as an exercise.

EXAMPLE:

```
sage: lp.get_backend().get_row_stat(0)
1
sage: lp.get_backend().get_col_stat(0)
1
```

Below we test that integers that can be exactly represented by IEEE 754 double-precision floating point numbers survive the rational reconstruction done by `glp_exact` and the subsequent conversion to double-precision floating point numbers.

EXAMPLE:

```
sage: lp = MixedIntegerLinearProgram(solver = 'GLPK', maximization = True)
sage: test = 2^53 - 43
sage: lp.solver_parameter("simplex_or_intopt", "exact_simplex_only") # use exact simplex onl
sage: x = lp[0]
sage: lp.add_constraint(x <= test)
sage: lp.set_objective(x)
sage: lp.solve() == test # yes, we want an exact comparison here
glp_exact: 1 rows, 1 columns, 1 non-zeros
GNU MP bignum library is being used
...
OPTIMAL SOLUTION FOUND
True
sage: lp.get_values(x) == test # yes, we want an exact comparison here
True
```

Below we test that GLPK backend can detect unboundedness in "simplex_only" mode (trac ticket #18838).

EXAMPLES:

```
sage: lp = MixedIntegerLinearProgram(maximization=True, solver = "GLPK")
sage: lp.set_objective(lp[0])
sage: lp.solver_parameter("simplex_or_intopt", "simplex_only")
sage: lp.solve()
Traceback (most recent call last):
...
MIPSolverException: 'GLPK : Problem has unbounded solution'
sage: lp.set_objective(lp[1])
sage: lp.solver_parameter("primal_v_dual", "GLP_DUAL")
sage: lp.solve()
Traceback (most recent call last):
...
MIPSolverException: 'GLPK : Problem has unbounded solution'
sage: lp.solver_parameter("simplex_or_intopt", "simplex_then_intopt")
sage: lp.solve()
Traceback (most recent call last):
...
MIPSolverException: 'GLPK : The LP (relaxation) problem has no dual feasible solution'
sage: lp.solver_parameter("simplex_or_intopt", "intopt_only")
sage: lp.solve()
Traceback (most recent call last):
...
MIPSolverException: 'GLPK : The LP (relaxation) problem has no dual feasible solution'
sage: lp.set_max(lp[1],5)
sage: lp.solve()
5.0
```

Solving a LP within the acceptable gap. No exception is raised, even if the result is not optimal. To do this, we try to compute the maximum number of disjoint balls (of diameter 1) in a hypercube:

```
sage: g = graphs.CubeGraph(9)
sage: p = MixedIntegerLinearProgram(solver="GLPK")
sage: p.solver_parameter("mip_gap_tolerance",100)
sage: b = p.new_variable(binary=True)
sage: p.set_objective(p.sum(b[v] for v in g))
sage: for v in g:
....:     p.add_constraint(b[v]+p.sum(b[u] for u in g.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1) # Force an easy non-0 solution
sage: p.solve() # rel tol 100
1
```

Same, now with a time limit:

```
sage: p.solver_parameter("mip_gap_tolerance",1)
sage: p.solver_parameter("timelimit",0.01)
sage: p.solve() # rel tol 1
1
```

solver_parameter (*name*, *value=None*)

Return or define a solver parameter

INPUT:

- `name` (string) – the parameter
- `value` – the parameter's value if it is to be defined, or `None` (default) to obtain its current value.

You can supply the name of a parameter and its value using either a string or a `glp_` constant (which are defined as Cython variables of this module).

In most cases, you can use the same name for a parameter as that given in the GLPK documentation, which is available by downloading GLPK from <http://www.gnu.org/software/glpk/>. The exceptions relate to parameters common to both methods; these require you to append `_simplex` or `_intopt` to the name to resolve ambiguity, since the interface allows access to both.

We have also provided more meaningful names, to assist readability.

Parameter **names** are specified in lower case. To use a constant instead of a string, prepend `glp_` to the name. For example, both `glp_gmi_cuts` or `"gmi_cuts"` control whether to solve using Gomory cuts.

Parameter **values** are specificed as strings in upper case, or as constants in lower case. For example, both `glp_on` and `"GLP_ON"` specify the same thing.

Naturally, you can use `True` and `False` in cases where `glp_on` and `glp_off` would be used.

A list of parameter names, with their possible values:

General-purpose parameters:

`timelimit`	specify the time limit IN SECONDS. This affects both simplex and intopt.
`timelimit_simplex` and `timelimit_intopt`	specify the time limit IN MILLISECONDS. (This is glpk's default.)
`simplex_or_intopt`	specifiy which of `simplex`, `exact` and `intopt` routines in GLPK to use. This is controlled by setting `simplex_or_intopt` to `glp_simplex_only`, `glp_exact_simplex_only`, `glp_intopt_only` and `glp_simplex_then_intopt`, respectively. The latter is useful to deal with a problem in GLPK where problems with no solution hang when using integer optimization; if you specify `glp_simplex_then_intopt`, sage will try simplex first, then perform integer optimization only if a solution of the LP relaxation exists.
`verbosity_intopt` and `verbosity_simplex`	one of `GLP_MSG_OFF`, `GLP_MSG_ERR`, `GLP_MSG_ON`, or `GLP_MSG_ALL`. The default is `GLP_MSG_OFF`.
`output_frequency_intopt` and `output_frequency_simplex`	the output frequency, in milliseconds. Default is 5000.
`output_delay_intopt` and `output_delay_simplex`	the output delay, in milliseconds, regarding the use of the simplex method on the LP relaxation. Default is 10000.

intopt-specific parameters:

`branching`	• `GLP_BR_FFV` first fractional variable • `GLP_BR_LFV` last fractional variable • `GLP_BR_MFV` most fractional variable • `GLP_BR_DTH` Driebeck-Tomlin heuristic (default) • `GLP_BR_PCH` hybrid pseudocost heuristic
`backtracking`	• `GLP_BT_DFS` depth first search • `GLP_BT_BFS` breadth first search • `GLP_BT_BLB` best local bound (default) • `GLP_BT_BPH` best projection heuristic
`preprocessing`	• `GLP_PP_NONE` • `GLP_PP_ROOT` preprocessing only at root level • `GLP_PP_ALL` (default)
`feasibility_pump`	`GLP_ON` or `GLP_OFF` (default)
`gomory_cuts`	`GLP_ON` or `GLP_OFF` (default)
`mixed_int_rounding_cuts`	`GLP_ON` or `GLP_OFF` (default)
`mixed_cover_cuts`	`GLP_ON` or `GLP_OFF` (default)
`clique_cuts`	`GLP_ON` or `GLP_OFF` (default)
`absolute_tolerance`	(double) used to check if optimal solution to LP relaxation is integer feasible. GLPK manual advises, "do not change... without detailed understanding of its purpose."
`relative_tolerance`	(double) used to check if objective value in LP relaxation is not better than best known integer solution. GLPK manual advises, "do not change... without detailed understanding of its purpose."
`mip_gap_tolerance`	(double) relative mip gap tolerance. Default is 0.0.
`presolve_intopt`	`GLP_ON` (default) or `GLP_OFF`.
`binarize`	`GLP_ON` or `GLP_OFF` (default)

simplex-specific parameters:

primal_v_dual	•GLP_PRIMAL (default) •GLP_DUAL •GLP_DUALP
pricing	•GLP_PT_STD standard (textbook) •GLP_PT_PSE projected steepest edge (default)
ratio_test	•GLP_RT_STD standard (textbook) •GLP_RT_HAR Harris' two-pass ratio test (default)
tolerance_primal	(double) tolerance used to check if basic solution is primal feasible. GLPK manual advises, "do not change... without detailed understanding of its purpose."
tolerance_dual	(double) tolerance used to check if basic solution is dual feasible. GLPK manual advises, "do not change... without detailed understanding of its purpose."
tolerance_pivot	(double) tolerance used to choose pivot. GLPK manual advises, "do not change... without detailed understanding of its purpose."
obj_lower_limit	(double) lower limit of the objective function. The default is -DBL_MAX.
obj_upper_limit	(double) upper limit of the objective function. The default is DBL_MAX.
iteration_limit	(int) iteration limit of the simplex algorithm. The default is INT_MAX.
presolve_simplex	GLP_ON or GLP_OFF (default).

Note: The coverage for GLPK's control parameters for simplex and integer optimization is nearly complete. The only thing lacking is a wrapper for callback routines.

To date, no attempt has been made to expose the interior point methods.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.solver_parameter("timelimit", 60)
sage: p.solver_parameter("timelimit")
60.0
```

- Don't forget the difference between timelimit and timelimit_intopt

```
sage: p.solver_parameter("timelimit_intopt")
60000
```

If you don't care for an integer answer, you can ask for an LP relaxation instead. The default solver performs integer optimization, but you can switch to the standard simplex algorithm through the glp_simplex_or_intopt parameter.

EXAMPLE:

```
sage: lp = MixedIntegerLinearProgram(solver = 'GLPK', maximization = False)
sage: x, y = lp[0], lp[1]
sage: lp.add_constraint(-2*x + y <= 1)
sage: lp.add_constraint(x - y <= 1)
sage: lp.add_constraint(x + y >= 2)
sage: lp.set_integer(x); lp.set_integer(y)
sage: lp.set_objective(x + y)
sage: lp.solve()
2.0
sage: lp.get_values([x,y])
[1.0, 1.0]
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
2.0
sage: lp.get_values([x,y])
[1.5, 0.5]
```

You can get GLPK to spout all sorts of information at you. The default is to turn this off, but sometimes (debugging) it's very useful:

```
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_then_intopt)
sage: lp.solver_parameter(backend.glp_mir_cuts, backend.glp_on)
sage: lp.solver_parameter(backend.glp_msg_lev_intopt, backend.glp_msg_all)
sage: lp.solver_parameter(backend.glp_mir_cuts)
1
```

If you actually try to solve `lp`, you will get a lot of detailed information.

variable_lower_bound(*index*, *value=False*)

Return or define the lower bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not lower bound. When set to `False` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_lower_bound(0, 5)
sage: p.col_bounds(0)
(5.0, None)
```

TESTS:

trac ticket #14581:

```
sage: P = MixedIntegerLinearProgram(solver="GLPK")
sage: x = P["x"]
sage: P.set_min(x, 5)
sage: P.set_min(x, 0)
sage: P.get_min(x)
0.0
```

variable_upper_bound(*index*, *value=False*)

Return or define the upper bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not upper bound. When set to `False` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0.0, 5.0)
```

TESTS:

trac ticket #14581:

```
sage: P = MixedIntegerLinearProgram(solver="GLPK")
sage: x = P["x"]
sage: P.set_max(x, 0)
sage: P.get_max(x)
0.0
```

warm_up()

Warm up the basis using current statuses assigned to rows and cols.

OUTPUT:

- Returns the warming up status
  - 0 The operation has been successfully performed.
  - GLP_EBADB The basis matrix is invalid.
  - GLP_ESING The basis matrix is singular within the working precision.
  - GLP_ECOND The basis matrix is ill-conditioned.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
sage: lp.add_linear_constraint(zip([0, 1, 2], [8, 6, 1]), None, 48)
sage: lp.add_linear_constraint(zip([0, 1, 2], [4, 2, 1.5]), None, 20)
sage: lp.add_linear_constraint(zip([0, 1, 2], [2, 1.5, 0.5]), None, 8)
sage: lp.set_objective([60, 30, 20])
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solve()
0
sage: lp.get_objective_value()
```

```
280.0
sage: lp.set_row_stat(0,3)
sage: lp.set_col_stat(1,1)
sage: lp.warm_up()
0
```

write_lp(*filename*)

Write the problem to a .lp file

INPUT:

- `filename` (string)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
sage: p.set_objective([2, 5])
sage: p.write_lp(os.path.join(SAGE_TMP, "lp_problem.lp"))
Writing problem data to ...
9 lines were written
```

write_mps(*filename*, *modern*)

Write the problem to a .mps file

INPUT:

- `filename` (string)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
sage: p.set_objective([2, 5])
sage: p.write_mps(os.path.join(SAGE_TMP, "lp_problem.mps"), 2)
Writing problem data to...
17 records were written
```

10.3 GLPK Backend for access to GLPK graph functions

AUTHORS:

- Christian Kuper (2012-11): Initial implementation

10.3.1 Methods index

Graph creation and modification operations:

add_vertex()	Adds an isolated vertex to the graph.
add_vertices()	Adds vertices from an iterable container of vertices.
set_vertex_demand()	Sets the vertex parameters.
set_vertices_demand()	Sets the parameters of selected vertices.
get_vertex()	Returns a specific vertex as a dict Object.
get_vertices()	Returns a dictionary of the dictonaries associated to each vertex.
vertices()	Returns a list of all vertices.
delete_vertex()	Removes a vertex from the graph.
delete_vertices()	Removes vertices from the graph.
add_edge()	Adds an edge between vertices u and v.
add_edges()	Adds edges to the graph.
get_edge()	Returns an edge connecting two vertices.
edges()	Returns a list of all edges in the graph.
delete_edge()	Deletes an edge from the graph.
delete_edges()	Deletes edges from the graph.

Graph writing operations:

write_graph()	Writes the graph to a plain text file.
write_ccdata()	Writes the graph to a text file in DIMACS format.
write_mincost()	Writes the mincost flow problem data to a text file in DIMACS format.
write_maxflow()	Writes the maximum flow problem data to a text file in DIMACS format.

Network optimization operations:

mincost_okalg()	Finds solution to the mincost problem with the out-of-kilter algorithm.
maxflow_ffalg()	Finds solution to the maxflow problem with Ford-Fulkerson algorithm.
cpp()	Solves the critical path problem of a project network.

10.3.2 Classes and methods

class sage.numerical.backends.glpk_graph_backend.**GLPKGraphBackend**

Bases: object

GLPK Backend for access to GLPK graph functions

The constructor can either be called without arguments (which results in an empty graph) or with arguments to read graph data from a file.

INPUT:

- data – a filename or a Graph object.
- format – when data is a filename, specifies the format of the data read from a file. The format parameter is a string and can take values as described in the table below.

Format parameters:

`plain`	Read data from a plain text file containing the following information: nv na i[1] j[1] i[2] j[2] . . . i[na] j[na] where: •nv is the number of vertices (nodes); •na is the number of arcs; •i[k], k = 1, . . . , na, is the index of tail vertex of arc k; •j[k], k = 1, . . . , na, is the index of head vertex of arc k.
`dimacs`	Read data from a plain ASCII text file in DIMACS format. A discription of the DIMACS format can be found at http://dimacs.rutgers.edu/Challenges/.
`mincost`	Reads the mincost flow problem data from a text file in DIMACS format
`maxflow`	Reads the maximum flow problem data from a text file in DIMACS format

Note: When `data` is a `Graph`, the following restrictions are applied.

- vertices – the value of the demand of each vertex (see `set_vertex_demand()`) is obtained from the numerical value associated with the key "rhs" if it is a dictionary.
- edges – The edge values used in the algorithms are read from the edges labels (and left undefined if the edge labels are equal to `None`). To be defined, the labels must be `dict` objects with keys "low", "cap" and "cost". See `get_edge()` for details.

EXAMPLES:

The following example creates an empty graph:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
```

The following example creates an empty graph, adds some data, saves the data to a file and loads it:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_vertices([None, None])
['0', '1']
sage: a = gbe.add_edge('0', '1')
sage: gbe.write_graph(SAGE_TMP+"/graph.txt")
Writing graph to ...
2 lines were written
0
sage: gbe1 = GLPKGraphBackend(SAGE_TMP+"/graph.txt", "plain")
Reading graph from ...
Graph has 2 vertices and 1 arc
2 lines were read
```

The following example imports a Sage `Graph` and then uses it to solve a maxflow problem:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: g = graphs.PappusGraph()
sage: for ed in g.edges():
...       g.set_edge_label(ed[0], ed[1], {"cap":1})
sage: gbe = GLPKGraphBackend(g)
sage: gbe.maxflow_ffalg('1', '2')
3.0
```

add_edge (*u*, *v*, *params=None*)

Adds an edge between vertices u and v.

Allows adding an edge and optionally providing parameters used by the algorithms. If a vertex does not exist it is created.

INPUT:

- u – The name (as str) of the tail vertex
- v – The name (as str) of the head vertex
- params – An optional dict containing the edge parameters used for the algorithms. The following keys are used:
  - low – The minimum flow through the edge
  - cap – The maximum capacity of the edge
  - cost – The cost of transporting one unit through the edge

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_edge("A", "B", {"low":0.0, "cap":10.0, "cost":5})
sage: gbe.vertices()
['A', 'B']
sage: for ed in gbe.edges():
...       print ed[0], ed[1], ed[2]['cap'], ed[2]['cost'], ed[2]['low']
A B 10.0 5.0 0.0
sage: gbe.add_edge("B", "C", {"low":0.0, "cap":10.0, "cost":'5'})
Traceback (most recent call last):
...
TypeError: Invalid edge parameter.
```

add_edges (*edges*)

Adds edges to the graph.

INPUT:

- edges – An iterable container of pairs of the form (u, v), where u is name (as str) of the tail vertex and v is the name (as str) of the head vertex or an interable container of triples of the form (u, v, params) where params is a dict as described in add_edge.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: edges = [("A", "B", {"low":0.0, "cap":10.0, "cost":5})]
sage: edges.append(("B", "C"))
sage: gbe.add_edges(edges)
sage: for ed in gbe.edges():
...       print ed[0], ed[1], ed[2]['cap'], ed[2]['cost'], ed[2]['low']
A B 10.0 5.0 0.0
```

```
B C 0.0 0.0 0.0
sage: edges = [("C", "D", {"low":0.0, "cap":10.0, "cost":5})]
sage: edges.append(("C", "E", 5))
sage: gbe.add_edges(edges)
Traceback (most recent call last):
...
TypeError: Argument 'params' has incorrect type ...
sage: for ed in gbe.edges():
...       print ed[0], ed[1], ed[2]['cap'], ed[2]['cost'], ed[2]['low']
A B 10.0 5.0 0.0
B C 0.0 0.0 0.0
C D 10.0 5.0 0.0
```

add_vertex(*name='NULL'*)

Adds an isolated vertex to the graph.

If the vertex already exists, nothing is done.

INPUT:

- **name – String of max 255 chars length. If no name is** specified, then the vertex will be represented by the string representation of the ID of the vertex or - if this already exists - a string representation of the least integer not already representing a vertex.

OUTPUT:

If no `name` is passed as an argument, the new vertex name is returned. `None` otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_vertex()
'0'
sage: gbe.add_vertex("2")
sage: gbe.add_vertex()
'1'
```

add_vertices(*vertices*)

Adds vertices from an iterable container of vertices.

Vertices that already exist in the graph will not be added again.

INPUT:

- `vertices` – iterator of vertex labels (`str`). A label can be `None`.

OUTPUT:

Generated names of new vertices if there is at least one `None` value present in `vertices`. `None` otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: vertices = [None for i in range(3)]
sage: gbe.add_vertices(vertices)
['0', '1', '2']
sage: gbe.add_vertices(['A', 'B', None])
['5']
sage: gbe.add_vertices(['A', 'B', 'C'])
```

```
sage: gbe.vertices()
['0', '1', '2', 'A', 'B', '5', 'C']
```

TESTS:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_vertices([None, None, None, '1'])
['0', '2', '3']
```

cpp()

Solves the critical path problem of a project network.

OUTPUT:

The length of the critical path of the network

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_vertices([None for i in range(3)])
['0', '1', '2']
sage: gbe.set_vertex_demand('0', 3)
sage: gbe.set_vertex_demand('1', 1)
sage: gbe.set_vertex_demand('2', 4)
sage: a = gbe.add_edge('0', '2')
sage: a = gbe.add_edge('1', '2')
sage: gbe.cpp()
7.0
sage: v = gbe.get_vertex('1')
sage: print 1, v["rhs"], v["es"], v["ls"] # abs tol 1e-6
1 1.0 0.0 2.0
```

delete_edge(*u*, *v*, *params=None*)

Deletes an edge from the graph.

If an edge does not exist it is ignored.

INPUT:

- `u` – The name (as `str`) of the tail vertex of the edge
- `v` – The name (as `str`) of the tail vertex of the edge
- `params` – `params` – An optional `dict` containing the edge parameters (see :meth:`add_edge`). If this parameter is not provided, all edges connecting `u` and `v` are deleted. Otherwise only edges with matching parameters are deleted.

See also:

`delete_edges()`

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: edges = [("A", "B", {"low":0.0, "cap":10.0, "cost":5})]
sage: edges.append(("A", "B", {"low":0.0, "cap":15.0, "cost":10}))
sage: edges.append(("B", "C", {"low":0.0, "cap":20.0, "cost":1}))
sage: edges.append(("B", "C", {"low":0.0, "cap":10.0, "cost":20}))
sage: gbe.add_edges(edges)
sage: gbe.delete_edge("A", "B")
```

```
sage: gbe.delete_edge("B", "C", {"low":0.0, "cap":10.0, "cost":20})
sage: print gbe.edges()[0][0], gbe.edges()[0][1], gbe.edges()[0][2]['cost']
B C 1.0
```

delete_edges (*edges*)

Deletes edges from the graph.

Non existing edges are ignored.

INPUT:

- `edges` – An iterable container of edges.

See also:

`delete_edge()`

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: edges = [("A", "B", {"low":0.0, "cap":10.0, "cost":5})]
sage: edges.append(("A", "B", {"low":0.0, "cap":15.0, "cost":10}))
sage: edges.append(("B", "C", {"low":0.0, "cap":20.0, "cost":1}))
sage: edges.append(("B", "C", {"low":0.0, "cap":10.0, "cost":20}))
sage: gbe.add_edges(edges)
sage: gbe.delete_edges(edges[1:])
sage: len(gbe.edges())
1
sage: print gbe.edges()[0][0], gbe.edges()[0][1], gbe.edges()[0][2]['cap']
A B 10.0
```

delete_vertex (*vert*)

Removes a vertex from the graph.

Trying to delete a non existing vertex will raise an exception.

INPUT:

- `vert` – The name (as `str`) of the vertex to delete.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: verts = ["A", "D"]
sage: gbe.add_vertices(verts)
sage: gbe.delete_vertex("A")
sage: gbe.vertices()
['D']
sage: gbe.delete_vertex("A")
Traceback (most recent call last):
...
RuntimeError: Vertex A does not exist.
```

delete_vertices (*verts*)

Removes vertices from the graph.

Trying to delete a non existing vertex will raise an exception.

INPUT:

- `verts` – iterable container containing names (as `str`) of the vertices to delete

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: verts = ["A", "B", "C", "D"]
sage: gbe.add_vertices(verts)
sage: v_d = ["A", "B"]
sage: gbe.delete_vertices(v_d)
sage: gbe.vertices()
['C', 'D']
sage: gbe.delete_vertices(["C", "A"])
Traceback (most recent call last):
...
RuntimeError: Vertex A does not exist.
sage: gbe.vertices()
['C', 'D']
```

edges()

Returns a `list` of all edges in the graph

OUTPUT:

A `list` of `triples` representing the edges of the graph.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: edges = [("A", "B", {"low":0.0, "cap":10.0, "cost":5})]
sage: edges.append(("B", "C"))
sage: gbe.add_edges(edges)
sage: for ed in gbe.edges():
...       print ed[0], ed[1], ed[2]['cost']
A B 5.0
B C 0.0
```

get_edge(*u*, *v*)

Returns an edge connecting two vertices.

Note: If multiple edges connect the two vertices only the first edge found is returned.

INPUT:

- `u` – Name (as `str`) of the tail vertex
- `v` – Name (as `str`) of the head vertex

OUTPUT:

A `triple` describing if edge was found or `None` if not. The third value of the triple is a `dict` containing the following edge parameters:

- `low` – The minimum flow through the edge
- `cap` – The maximum capacity of the edge
- `cost` – The cost of transporting one unit through the edge
- `x` – The actual flow through the edge after solving

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: edges = [("A", "B"), ("A", "C"), ("B", "C")]
sage: gbe.add_edges(edges)
sage: ed = gbe.get_edge("A", "B")
sage: print ed[0], ed[1], ed[2]['x']
A B 0.0
sage: gbe.get_edge("A", "F") is None
True
```

get_vertex(*vertex*)

Returns a specific vertex as a `dict` Object.

INPUT:

- `vertex` – The vertex label as `str`.

OUTPUT:

The vertex as a `dict` object or `None` if the vertex does not exist. The `dict` contains the values used or created by the different algorithms. The values associated with the keys following keys contain:

- "rhs" – The supply / demand value the vertex (mincost alg)
- "pi" – The node potential (mincost alg)
- "cut" – The cut flag of the vertex (maxflow alg)
- "es" – The earliest start of task (cpp alg)
- "ls" – The latest start of task (cpp alg)

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: verts = ["A", "B", "C", "D"]
sage: gbe.add_vertices(verts)
sage: sorted(gbe.get_vertex("A").items())
[('cut', 0), ('es', 0.0), ('ls', 0.0), ('pi', 0.0), ('rhs', 0.0)]
sage: gbe.get_vertex("F") is None
True
```

get_vertices(*verts*)

Returns a dictionary of the dictonaries associated to each vertex.

INPUT:

- `verts` – iterable container of vertices

OUTPUT:

A list of pairs `(vertex, properties)` where `properties` is a dictionary containing the numerical values associated with a vertex. For more information, see the documentation of `GLPKGraphBackend.get_vertex()`.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: verts = ['A', 'B']
sage: gbe.add_vertices(verts)
sage: sorted(gbe.get_vertices(verts)['B'].items())
[('cut', 0), ('es', 0.0), ('ls', 0.0), ('pi', 0.0), ('rhs', 0.0)]
```

```
sage: gbe.get_vertices(["C", "D"])
{}
```

maxflow_ffalg(*u=None*, *v=None*)

Finds solution to the maxflow problem with Ford-Fulkerson algorithm.

INPUT:

- u – Name (as str) of the tail vertex. Default is None.
- v – Name (as str) of the head vertex. Default is None.

If u or v are None, the currently stored values for the head or tail vertex are used. This behavior is useful when reading maxflow data from a file. When calling this function with values for u and v, the head and tail vertex are stored for later use.

OUTPUT:

The solution to the maxflow problem, i.e. the maximum flow.

Note:

- If the source or sink vertex does not exist, an IndexError is raised.
- If the source and sink are identical, a ValueError is raised.
- This method raises MIPSolverException exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: v = gbe.add_vertices([None for i in range(5)])
sage: edges = ((0, 1, 2), (0, 2, 3), (1, 2, 3), (1, 3, 4),
...           (3, 4, 1), (2, 4, 2))
sage: for a in edges:
...       edge = gbe.add_edge(str(a[0]), str(a[1]), {"cap":a[2]})
sage: gbe.maxflow_ffalg('0', '4')
3.0
sage: gbe.maxflow_ffalg()
3.0
sage: gbe.maxflow_ffalg('0', '8')
Traceback (most recent call last):
...
IndexError: Source or sink vertex does not exist
```

mincost_okalg()

Finds solution to the mincost problem with the out-of-kilter algorithm.

The out-of-kilter algorithm requires all problem data to be integer valued.

OUTPUT:

The solution to the mincost problem, i.e. the total cost, if operation was successful.

Note: This method raises MIPSolverException exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: vertices = (35, 50, 40, -45, -20, -30, -30)
sage: vs = gbe.add_vertices([None for i in range(len(vertices))])
sage: v_dict = {}
sage: for i, v in enumerate(vs):
...       v_dict[v] = vertices[i]
sage: gbe.set_vertices_demand(v_dict.items())
sage: cost = ((8, 6, 10, 9), (9, 12, 13, 7), (14, 9, 16, 5))
sage: lcost = range(len(cost))
sage: lcost_0 = range(len(cost[0]))
sage: for i in lcost:
...       for j in lcost_0:
...           gbe.add_edge(str(i), str(j + len(cost)), {"cost":cost[i][j], "cap":100})
sage: gbe.mincost_okalg()
1020.0
sage: for ed in gbe.edges():
...        print ed[0], "->", ed[1], ed[2]["x"]
0 -> 6 0.0
0 -> 5 25.0
0 -> 4 10.0
0 -> 3 0.0
1 -> 6 0.0
1 -> 5 5.0
1 -> 4 0.0
1 -> 3 45.0
2 -> 6 30.0
2 -> 5 0.0
2 -> 4 10.0
2 -> 3 0.0
```

set_vertex_demand (*vertex*, *demand*)

Sets the demand of the vertex in a mincost flow algorithm.

INPUT:

- `vertex` – Name of the vertex
- `demand` – the numerical value representing demand of the vertex in a mincost flow alorithm (it could be for instance -1 to represent a sink, or 1 to represent a source and 0 for a neutral vertex). This can either be an `int` or `float` value.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: vertices = [None for i in range(3)]
sage: gbe.add_vertices(vertices)
['0', '1', '2']
sage: gbe.set_vertex_demand('0', 2)
sage: gbe.get_vertex('0')['rhs']
2.0
sage: gbe.set_vertex_demand('3', 2)
Traceback (most recent call last):
...
KeyError: 'Vertex 3 does not exist.'
```

set_vertices_demand (*pairs*)

Sets the parameters of selected vertices.

INPUT:

- `pairs` – A list of pairs `(vertex, demand)` associating a demand to each vertex. For more information, see the documentation of `set_vertex_demand()`.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: vertices = [None for i in range(3)]
sage: gbe.add_vertices(vertices)
['0', '1', '2']
sage: gbe.set_vertices_demand([('0', 2), ('1', 3), ('3', 4)])
sage: sorted(gbe.get_vertex('1').items())
[('cut', 0), ('es', 0.0), ('ls', 0.0), ('pi', 0.0), ('rhs', 3.0)]
```

vertices()

Returns the list of all vertices

Note: Changing elements of the `list` will not change anything in the the graph.

Note: If a vertex in the graph does not have a name / label it will appear as `None` in the resulting `list`.

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: verts = ["A", "B", "C"]
sage: gbe.add_vertices(verts)
sage: a = gbe.vertices(); a
['A', 'B', 'C']
sage: a.pop(0)
'A'
sage: gbe.vertices()
['A', 'B', 'C']
```

write_ccdata(*fname*)

Writes the graph to a text file in DIMACS format.

Writes the data to plain ASCII text file in DIMACS format. A discription of the DIMACS format can be found at http://dimacs.rutgers.edu/Challenges/.

INPUT:

- `fname` – full name of the file

OUTPUT:

Zero if the operations was successful otherwise nonzero

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: a = gbe.add_edge("0", "1")
sage: gbe.write_ccdata(SAGE_TMP+"/graph.dat")
Writing graph to ...
6 lines were written
0
```

write_graph (*fname*)

Writes the graph to a plain text file

INPUT:

- `fname` – full name of the file

OUTPUT:

Zero if the operations was successful otherwise nonzero

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: a = gbe.add_edge("0", "1")
sage: gbe.write_graph(SAGE_TMP+"/graph.txt")
Writing graph to ...
2 lines were written
0
```

write_maxflow (*fname*)

Writes the maximum flow problem data to a text file in DIMACS format.

INPUT:

- `fname` – Full name of file

OUTPUT:

`Zero` if successful, otherwise `non-zero`

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: gbe.add_vertices([None for i in range(2)])
['0', '1']
sage: a = gbe.add_edge('0', '1')
sage: gbe.maxflow_ffalg('0', '1')
0.0
sage: gbe.write_maxflow(SAGE_TMP+"/graph.max")
Writing maximum flow problem data to ...
6 lines were written
0
sage: gbe = GLPKGraphBackend()
sage: gbe.write_maxflow(SAGE_TMP+"/graph.max")
Traceback (most recent call last):
...
IOError: Cannot write empty graph
```

write_mincost (*fname*)

Writes the mincost flow problem data to a text file in DIMACS format

INPUT:

- `fname` – Full name of file

OUTPUT:

Zero if successful, otherwise nonzero

EXAMPLE:

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: a = gbe.add_edge("0", "1")
sage: gbe.write_mincost(SAGE_TMP+"/graph.min")
Writing min-cost flow problem data to ...
4 lines were written
0
```

10.4 PPL Backend

AUTHORS:

- Risan (2012-02): initial implementation
- Jeroen Demeyer (2014-08-04) allow rational coefficients for constraints and objective function (trac ticket #16755)

class sage.numerical.backends.ppl_backend.**PPLBackend**

Bases: sage.numerical.backends.generic_backend.GenericBackend

add_col(*indices*, *coeffs*)

Add a column.

INPUT:

- indices (list of integers) – this list constains the indices of the constraints in which the variable's coefficient is nonzero
- coeffs (list of real values) – associates a coefficient to the variable in each of the constraints in which it appears. Namely, the ith entry of coeffs corresponds to the coefficient of the variable in the constraint represented by the ith entry in indices.

Note: indices and coeffs are expected to be of the same length.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.nrows()
0
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(range(5), range(5))
sage: p.nrows()
5
```

add_linear_constraint(*coefficients*, *lower_bound*, *upper_bound*, *name=None*)

Add a linear constraint.

INPUT:

- coefficients – an iterable with (c,v) pairs where c is a variable index (integer) and v is a value (real value).
- lower_bound – a lower bound, either a real value or None
- upper_bound – an upper bound, either a real value or None

- name – an optional name for this row (default: None)

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver="PPL")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(x[0]/2 + x[1]/3 <= 2/5)
sage: p.set_objective(x[1])
sage: p.solve()
6/5
sage: p.add_constraint(x[0] - x[1] >= 1/10)
sage: p.solve()
21/50
sage: p.set_max(x[0], 1/2)
sage: p.set_min(x[1], 3/8)
sage: p.solve()
2/5

sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2.0, 2.0)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2.00000000000000, 2.00000000000000)
sage: p.add_linear_constraint( zip(range(5), range(5)), 1.0, 1.0, name='foo')
sage: p.row_name(-1)
'foo'
```

add_linear_constraints(*number*, *lower_bound*, *upper_bound*, *names=None*)

Add constraints.

INPUT:

- number (integer) – the number of constraints to add.
- lower_bound – a lower bound, either a real value or None
- upper_bound – an upper bound, either a real value or None
- names – an optional list of names (default: None)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(5, None, 2)
sage: p.row(4)
([], [])
sage: p.row_bounds(4)
(None, 2)
```

add_variable(*lower_bound=0*, *upper_bound=None*, *binary=False*, *continuous=True*, *integer=False*, *obj=0*, *name=None*)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both positive and real.

It has not been implemented for selecting the variable type yet.

INPUT:

- `lower_bound` – the lower bound of the variable (default: 0)
- `upper_bound` – the upper bound of the variable (default: `None`)
- `binary` – `True` if the variable is binary (default: `False`).
- `continuous` – `True` if the variable is binary (default: `True`).
- `integer` – `True` if the variable is binary (default: `False`).
- `obj` – (optional) coefficient of this variable in the objective function (default: 0)
- `name` – an optional name for the newly added variable (default: `None`).

OUTPUT: The index of the newly created variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.ncols()
1
sage: p.add_variable(lower_bound=-2)
1
sage: p.add_variable(name='x',obj=2/3)
2
sage: p.col_name(2)
'x'
sage: p.objective_coefficient(2)
2/3
```

add_variables(*n*, *lower_bound=0*, *upper_bound=None*, *binary=False*, *continuous=True*, *integer=False*, *obj=0*, *names=None*)

Add `n` variables.

This amounts to adding new columns to the matrix. By default, the variables are both positive and real.

It has not been implemented for selecting the variable type yet.

INPUT:

- `n` – the number of new variables (must be > 0)
- `lower_bound` – the lower bound of the variable (default: 0)
- `upper_bound` – the upper bound of the variable (default: `None`)
- `binary` – `True` if the variable is binary (default: `False`).
- `continuous` – `True` if the variable is binary (default: `True`).
- `integer` – `True` if the variable is binary (default: `False`).
- `obj` – (optional) coefficient of all variables in the objective function (default: 0)
- `names` – optional list of names (default: `None`)

OUTPUT: The index of the variable created last.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variables(5)
4
sage: p.ncols()
5
sage: p.add_variables(2, lower_bound=-2.0, names=['a','b'])
6
```

base_ring()

col_bounds(*index*)

Return the bounds of a specific variable.

INPUT:

- `index` (integer) – the variable's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the variable is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0, 5)
```

col_name(*index*)

Return the `index` th col name

INPUT:

- `index` (integer) – the col's id
- `name` (`char *`) – its name. When set to `NULL` (default), the method returns the current name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variable(name="I am a variable")
0
sage: p.col_name(0)
'I am a variable'
```

get_objective_value()

Return the exact value of the objective function.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(solver="PPL")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(5/13*x[0] + x[1]/2 == 8/7)
sage: p.set_objective(5/13*x[0] + x[1]/2)
sage: p.solve()
8/7
```

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([(0,1), (1,2)], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: p.get_objective_value()
15/2
sage: p.get_variable_value(0)
0
sage: p.get_variable_value(1)
3/2
```

get_variable_value(*variable*)

Return the value of a variable given by the solver.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([(0,1), (1, 2)], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: p.get_objective_value()
15/2
sage: p.get_variable_value(0)
0
sage: p.get_variable_value(1)
3/2
```

init_mip()

Converting the matrix form of the MIP Problem to PPL MIP_Problem.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="PPL")
sage: p.base_ring()
Rational Field
sage: type(p.zero())
<type 'sage.rings.rational.Rational'>
sage: p.init_mip()
```

is_maximization()

Test whether the problem is a maximization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

is_variable_binary (*index*)

Test whether the given variable is of binary type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_binary(0)
False
```

is_variable_continuous (*index*)

Test whether the given variable is of continuous/real type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_continuous(0)
True
```

is_variable_integer (*index*)

Test whether the given variable is of integer type.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variable()
0
```

```
sage: p.is_variable_integer(0)
False
```

ncols()

Return the number of columns/variables.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.nrows()
0
sage: p.add_linear_constraints(2, 2.0, None)
sage: p.nrows()
2
```

objective_coefficient(*variable*, *coeff=None*)

Set or get the coefficient of a variable in the objective function

INPUT:

- `variable` (integer) – the variable's id
- `coeff` (integer) – its coefficient

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0
sage: p.objective_coefficient(0,2)
sage: p.objective_coefficient(0)
2
```

problem_name(*name='NULL'*)

Return or define the problem's name

INPUT:

- `name` (`char *`) – the problem's name. When set to `NULL` (default), the method returns the problem's name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.problem_name("There once was a french fry")
sage: print p.problem_name()
There once was a french fry
```

row (*i*)

Return a row

INPUT:

•index (integer) – the constraint's id.

OUTPUT:

A pair (indices, coeffs) where indices lists the entries whose coefficient is nonzero, and to which coeffs associates their coefficient on the model of the add_linear_constraint method.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2, 2)
```

row_bounds (*index*)

Return the bounds of a specific constraint.

INPUT:

•index (integer) – the constraint's id.

OUTPUT:

A pair (lower_bound, upper_bound). Each of them can be set to None if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2, 2)
```

row_name (*index*)

Return the index th row name

INPUT:

•index (integer) – the row's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_linear_constraints(1, 2, None, names="Empty constraint 1")
sage: p.row_name(0)
'Empty constraint 1'
```

set_objective (*coeff*, *d=0*)

Set the objective function.

INPUT:

- `coeff` – a list of real values, whose ith element is the coefficient of the ith variable in the objective function.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(solver="PPL")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(x[0]*5 + x[1]/11 <= 6)
sage: p.set_objective(x[0])
sage: p.solve()
6/5
sage: p.set_objective(x[0]/2 + 1)
sage: p.show()
Maximization:
  1/2 x_0 + 1

Constraints:
  constraint_0: 5 x_0 + 1/11 x_1 <= 6
Variables:
  x_0 is a continuous variable (min=0, max=+oo)
  x_1 is a continuous variable (min=0, max=+oo)
sage: p.solve()
8/5
```

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
sage: map(lambda x :p.objective_coefficient(x), range(5))
[1, 1, 2, 1, 3]
```

set_sense (*sense*)

Set the direction (maximization/minimization).

INPUT:

- `sense` (integer) :
  - +1 => Maximization
  - −1 => Minimization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
```

```
sage: p.is_maximization()
False
```

set_variable_type (*variable, vtype*)

Set the type of a variable.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(5)
4
sage: p.set_variable_type(3, -1)
sage: p.set_variable_type(3, -2)
Traceback (most recent call last):
...
Exception: ...
```

set_verbosity (*level*)

Set the log (verbosity) level. Not Implemented.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.set_verbosity(0)
```

solve ()

Solve the problem.

Note: This method raises `MIPSolverException` exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(range(5), range(5))
sage: p.solve()
0
sage: p.objective_coefficient(0,1)
sage: p.solve()
Traceback (most recent call last):
...
MIPSolverException: ...
```

variable_lower_bound (*index, value=False*)

Return or define the lower bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not lower bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0, None)
sage: p.variable_lower_bound(0, 5)
sage: p.col_bounds(0)
(5, None)
sage: p.variable_lower_bound(0, None)
sage: p.col_bounds(0)
(None, None)
```

variable_upper_bound(*index*, *value=False*)

Return or define the upper bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not upper bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0, 5)
sage: p.variable_upper_bound(0, None)
sage: p.col_bounds(0)
(0, None)
```

zero()

10.5 UNABLE TO IMPORT MODULE

AUTHORS:

- Ingolfur Edvardsson (2014-05) : initial implementation

class sage.numerical.backends.cvxopt_backend.**CVXOPTBackend**

Bases: `sage.numerical.backends.generic_backend.GenericBackend`

add_col(*indices*, *coeffs*)

Add a column.

INPUT:

- `indices` (list of integers) – this list constains the indices of the constraints in which the variable's coefficient is nonzero

- `coeffs` (list of real values) – associates a coefficient to the variable in each of the constraints in which it appears. Namely, the ith entry of `coeffs` corresponds to the coefficient of the variable in the constraint represented by the ith entry in `indices`.

Note: `indices` and `coeffs` are expected to be of the same length.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.nrows()
0
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(range(5), range(5))
sage: p.nrows()
5
```

add_linear_constraint (*coefficients*, *lower_bound*, *upper_bound*, *name=None*)

Add a linear constraint.

INPUT:

- `coefficients` an iterable with `(c,v)` pairs where `c` is a variable index (integer) and `v` is a value (real value).
- `lower_bound` - a lower bound, either a real value or `None`
- `upper_bound` - an upper bound, either a real value or `None`
- `name` - an optional name for this row (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2.0, 2.0)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2.00000000000000, 2.00000000000000)
sage: p.add_linear_constraint( zip(range(5), range(5)), 1.0, 1.0, name='foo')
sage: p.row_name(-1)
'foo'
```

add_linear_constraints (*number*, *lower_bound*, *upper_bound*, *names=None*)

Add constraints.

INPUT:

- `number` (integer) – the number of constraints to add.
- `lower_bound` - a lower bound, either a real value or `None`
- `upper_bound` - an upper bound, either a real value or `None`
- `names` - an optional list of names (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(5, None, 2)
sage: p.row(4)
([], [])
sage: p.row_bounds(4)
(None, 2)
```

add_variable(*lower_bound=0.0, upper_bound=None, binary=False, continuous=True, integer=False, obj=None, name=None*)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both positive and real. Variable types are always continuous, and thus the parameters `binary`, `integer`, and `continuous` have no effect.

INPUT:

- `lower_bound` - the lower bound of the variable (default: 0)
- `upper_bound` - the upper bound of the variable (default: `None`)
- `binary` - `True` if the variable is binary (default: `False`).
- `continuous` - `True` if the variable is continuous (default: `True`).
- `integer` - `True` if the variable is integer (default: `False`).
- `obj` - (optional) coefficient of this variable in the objective function (default: 0.0)
- `name` - an optional name for the newly added variable (default: `None`).

OUTPUT: The index of the newly created variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.ncols()
1
sage: p.add_variable()
1
sage: p.add_variable(lower_bound=-2.0)
2
sage: p.add_variable(continuous=True)
3
sage: p.add_variable(name='x',obj=1.0)
4
sage: p.col_name(3)
'x_3'
sage: p.col_name(4)
'x'
sage: p.objective_coefficient(4)
1.00000000000000
```

TESTS:

```
sage: p.add_variable(integer=True)
Traceback (most recent call last):
...
RuntimeError: CVXOPT only supports continuous variables
sage: p.add_variable(binary=True)
Traceback (most recent call last):
...
RuntimeError: CVXOPT only supports continuous variables
```

add_variables (*n*, *lower_bound=None*, *upper_bound=None*, *binary=False*, *continuous=True*, *integer=False*, *obj=None*, *names=None*)

Add `n` variables.

This amounts to adding new columns to the matrix. By default, the variables are both positive and real.

INPUT:

- `n` - the number of new variables (must be > 0)
- `lower_bound` - the lower bound of the variable (default: 0)
- `upper_bound` - the upper bound of the variable (default: `None`)
- `binary` - `True` if the variable is binary (default: `False`).
- `continuous` - `True` if the variable is binary (default: `True`).
- `integer` - `True` if the variable is binary (default: `False`).
- `obj` - (optional) coefficient of all variables in the objective function (default: 0.0)
- `names` - optional list of names (default: `None`)

OUTPUT: The index of the variable created last.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variables(5)
4
sage: p.ncols()
5
sage: p.add_variables(2, lower_bound=-2.0, integer=True, names=['a','b'])
6
```

col_bounds (*index*)

Return the bounds of a specific variable.

INPUT:

- `index` (integer) – the variable's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the variable is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable()
```

```
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0.0, 5)
```

col_name(*index*)

Return the `index` th col name

INPUT:

- `index` (integer) – the col's id
- `name` (`char *`) – its name. When set to `NULL` (default), the method returns the current name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable(name="I am a variable")
0
sage: p.col_name(0)
'I am a variable'
```

get_objective_value()

Return the value of the objective function.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "cvxopt")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([(0,1), (1,2)], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: round(p.get_objective_value(),4)
7.5
sage: round(p.get_variable_value(0),4)
0.0
sage: round(p.get_variable_value(1),4)
1.5
```

get_variable_value(*variable*)

Return the value of a variable given by the solver.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([(0,1), (1, 2)], None, 3)
```

```
sage: p.set_objective([2, 5])
sage: p.solve()
0
sage: round(p.get_objective_value(),4)
7.5
sage: round(p.get_variable_value(0),4)
0.0
sage: round(p.get_variable_value(1),4)
1.5
```

is_maximization()

Test whether the problem is a maximization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

is_variable_binary(*index*)

Test whether the given variable is of binary type. CVXOPT does not allow integer variables, so this is a bit moot.

INPUT:

•`index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,0)
Traceback (most recent call last):
...
ValueError: ...
sage: p.is_variable_binary(0)
False
```

is_variable_continuous(*index*)

Test whether the given variable is of continuous/real type. CVXOPT does not allow integer variables, so this is a bit moot.

INPUT:

•`index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variable()
```

```
0
sage: p.is_variable_continuous(0)
True
sage: p.set_variable_type(0,1)
Traceback (most recent call last):
...
ValueError: ...
sage: p.is_variable_continuous(0)
True
```

is_variable_integer(*index*)

Test whether the given variable is of integer type. CVXOPT does not allow integer variables, so this is a bit moot.

INPUT:

- `index` (integer) – the variable's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,-1)
sage: p.set_variable_type(0,1)
Traceback (most recent call last):
...
ValueError: ...
sage: p.is_variable_integer(0)
False
```

ncols()

Return the number of columns/variables.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.nrows()
0
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(2, 2.0, None)
```

```
sage: p.nrows()
2
```

objective_coefficient (*variable*, *coeff=None*)

Set or get the coefficient of a variable in the objective function

INPUT:

- `variable` (integer) – the variable's id
- `coeff` (double) – its coefficient

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0.0
sage: p.objective_coefficient(0,2)
sage: p.objective_coefficient(0)
2.0
```

problem_name (*name='NULL'*)

Return or define the problem's name

INPUT:

- `name` (`char *`) – the problem's name. When set to `NULL` (default), the method returns the problem's name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.problem_name("There once was a french fry")
sage: print p.problem_name()
There once was a french fry
```

row (*i*)

Return a row

INPUT:

- `index` (integer) – the constraint's id.

OUTPUT:

A pair `(indices, coeffs)` where `indices` lists the entries whose coefficient is nonzero, and to which `coeffs` associates their coefficient on the model of the `add_linear_constraint` method.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2, 2)
```

row_bounds (*index*)

Return the bounds of a specific constraint.

INPUT:

- `index` (integer) – the constraint's id.

OUTPUT:

A pair `(lower_bound, upper_bound)`. Each of them can be set to `None` if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
sage: p.row_bounds(0)
(2, 2)
```

row_name (*index*)

Return the `index` th row name

INPUT:

- `index` (integer) – the row's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_linear_constraints(1, 2, None, names=["Empty constraint 1"])
sage: p.row_name(0)
'Empty constraint 1'
```

set_objective (*coeff*, *d=0.0*)

Set the objective function.

INPUT:

- `coeff` – a list of real values, whose ith element is the coefficient of the ith variable in the objective function.
- `d` (double) – the constant term in the linear function (set to 0 by default)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
sage: map(lambda x :p.objective_coefficient(x), range(5))
[1, 1, 2, 1, 3]
```

set_sense (*sense*)

Set the direction (maximization/minimization).

INPUT:

- sense (integer) :
  - +1 => Maximization
  - -1 => Minimization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

set_variable_type (*variable*, *vtype*)
Set the type of a variable.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "cvxopt")
sage: p.add_variables(5)
4
sage: p.set_variable_type(3, -1)
sage: p.set_variable_type(3, -2)
Traceback (most recent call last):
...
ValueError: ...
```

set_verbosity (*level*)
Does not apply for the cvxopt solver

solve ()
Solve the problem.

Note: This method raises MIPSolverException exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: p = MixedIntegerLinearProgram(solver = "cvxopt", maximization=False)
sage: x=p.new_variable(nonnegative=True)
sage: p.set_objective(-4*x[0] - 5*x[1])
sage: p.add_constraint(2*x[0] + x[1] <= 3)
sage: p.add_constraint(2*x[1] + x[0] <= 3)
sage: round(p.solve(), 2)
-9.0
sage: p = MixedIntegerLinearProgram(solver = "cvxopt", maximization=False)
sage: x=p.new_variable(nonnegative=True)
sage: p.set_objective(x[0] + 2*x[1])
sage: p.add_constraint(-5*x[0] + x[1]   <=    7)
sage: p.add_constraint(-5*x[0] + x[1]   >=    7)
sage: p.add_constraint(x[0] + x[1] >= 26  )
sage: p.add_constraint( x[0] >= 3)
sage: p.add_constraint( x[1] >= 4)
sage: round(p.solve(),2)
48.83
sage: p = MixedIntegerLinearProgram(solver = "cvxopt")
sage: x=p.new_variable(nonnegative=True)
```

```
sage: p.set_objective(x[0] + x[1] + 3*x[2])
sage: p.solver_parameter("show_progress",True)
sage: p.add_constraint(x[0] + 2*x[1] <= 4)
sage: p.add_constraint(5*x[2] - x[1] <= 8)
sage: round(p.solve(), 2)
...
         pcost       dcost       gap    pres   dres   k/t
    ...
   8.8
sage: #CVXOPT gives different  values for variables compared to the other solvers.
sage: c = MixedIntegerLinearProgram(solver = "cvxopt")
sage: p = MixedIntegerLinearProgram(solver = "ppl")
sage: g = MixedIntegerLinearProgram()
sage: xc=c.new_variable(nonnegative=True)
sage: xp=p.new_variable(nonnegative=True)
sage: xg=g.new_variable(nonnegative=True)
sage: c.set_objective(xc[2])
sage: p.set_objective(xp[2])
sage: g.set_objective(xg[2])
sage: #we create a cube for all three solvers
sage: c.add_constraint(xc[0] <= 100)
sage: c.add_constraint(xc[1] <= 100)
sage: c.add_constraint(xc[2] <= 100)
sage: p.add_constraint(xp[0] <= 100)
sage: p.add_constraint(xp[1] <= 100)
sage: p.add_constraint(xp[2] <= 100)
sage: g.add_constraint(xg[0] <= 100)
sage: g.add_constraint(xg[1] <= 100)
sage: g.add_constraint(xg[2] <= 100)
sage: round(c.solve(),2)
100.0
sage: round(c.get_values(xc[0]),2)
50.0
sage: round(c.get_values(xc[1]),2)
50.0
sage: round(c.get_values(xc[2]),2)
100.0
sage: round(p.solve(),2)
100.0
sage: round(p.get_values(xp[0]),2)
0.0
sage: round(p.get_values(xp[1]),2)
0.0
sage: round(p.get_values(xp[2]),2)
100.0
sage: round(g.solve(),2)
100.0
sage: round(g.get_values(xg[0]),2)
0.0
sage: round(g.get_values(xg[1]),2)
0.0
sage: round(g.get_values(xg[2]),2)
100.0
```

solver_parameter (*name*, *value=None*)

Return or define a solver parameter

INPUT:

- `name` (string) – the parameter
- `value` – the parameter's value if it is to be defined, or `None` (default) to obtain its current value.

Note: The list of available parameters is available at `solver_parameter()`.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.solver_parameter("show_progress")
False
sage: p.solver_parameter("show_progress", True)
sage: p.solver_parameter("show_progress")
True
```

variable_lower_bound(*index*, *value=None*)

Return or define the lower bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not lower bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver

sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_lower_bound(0, 5)
sage: p.col_bounds(0)
(5, None)
```

variable_upper_bound(*index*, *value=None*)

Return or define the upper bound on a variable

INPUT:

- `index` (integer) – the variable's id
- `value` – real value, or `None` to mean that the variable has not upper bound. When set to `None` (default), the method returns the current value.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable()
0
sage: p.col_bounds(0)
(0.0, None)
sage: p.variable_upper_bound(0, 5)
sage: p.col_bounds(0)
(0.0, 5)
```

Sage also supports, via optional packages, CBC (COIN-OR), CPLEX (ILOG), and Gurobi. In order to find out how to use them in Sage, please refer to the Thematic Tutorial on Linear Programming.

CHAPTER

ELEVEN

SEMIDEFINITE OPTIMIZATION (SDP) SOLVER BACKENDS

11.1 UNABLE TO IMPORT MODULE

AUTHORS:

- Ingolfur Edvardsson (2014-05) : initial implementation
- Dima Pasechnik (2015-12) : minor fixes

class `sage.numerical.backends.cvxopt_sdp_backend.`**`CVXOPTSDPBackend`**

Bases: `sage.numerical.backends.generic_sdp_backend.GenericSDPBackend`

`add_linear_constraint`(*coefficients*, *name=None*)

Add a linear constraint.

INPUT:

- `coefficients` an iterable with `(c,v)` pairs where `c` is a variable index (integer) and `v` is a value (real value). The pairs come sorted by indices. If c is -1 it represents the constant coefficient.
- `name` - an optional name for this row (default: `None`)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint(  [(0, matrix([[33., -9.], [-9., 26.]])) , (1,  matrix([[-7.,
sage: p.row(0)
([0, 1],
 [
[ 33.0000000000000 -9.00000000000000]
[-9.00000000000000  26.0000000000000],

[-7.00000000000000 -11.0000000000000]
[-11.0000000000000  3.00000000000000]
])
sage: p.add_linear_constraint(  [(0, matrix([[33., -9.], [-9., 26.]])) , (1,  matrix([[-7.,
sage: p.row_name(-1)
'fun'
```

`add_linear_constraints`(*number*, *names=None*)

Add constraints.

INPUT:

- `number` (integer) – the number of constraints to add.

- names - an optional list of names (default: None)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(5)
sage: p.row(4)
([], [])
```

add_variable(*obj=0.0*, *name=None*)

Add a variable.

This amounts to adding a new column of matrices to the matrix. By default, the variable is both positive and real.

INPUT:

- obj - (optional) coefficient of this variable in the objective function (default: 0.0)
- name - an optional name for the newly added variable (default: None).

OUTPUT: The index of the newly created variable

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.ncols()
1
sage: p.add_variable()
1
sage: p.add_variable(name='x',obj=1.0)
2
sage: p.col_name(2)
'x'
sage: p.objective_coefficient(2)
1.00000000000000
```

add_variables(*n*, *names=None*)

Add n variables.

This amounts to adding new columns to the matrix. By default, the variables are both positive and real.

INPUT:

- n - the number of new variables (must be > 0)
- names - optional list of names (default: None)

OUTPUT: The index of the variable created last.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
```

```
sage: p.add_variables(5)
4
sage: p.ncols()
5
sage: p.add_variables(2, names=['a','b'])
6
```

col_name (*index*)

Return the `index` th col name

INPUT:

- `index` (integer) – the col's id
- `name` (`char *`) – its name. When set to `NULL` (default), the method returns the current name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable(name="I am a variable")
0
sage: p.col_name(0)
'I am a variable'
```

get_matrix ()

Get a block of a matrix coefficient

EXAMPLE:

```
sage: p = SemidefiniteProgram(solver="cvxopt")
sage: x = p.new_variable()
sage: a1 = matrix([[1, 2.], [2., 3.]])
sage: a2 = matrix([[3, 4.], [4., 5.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] <= a1)
sage: b = p.get_backend()
sage: b.get_matrix()[0][0]
(
    [-1.0 -2.0]
-1, [-2.0 -3.0]
)
```

get_objective_value ()

Return the value of the objective function.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "cvxopt")
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1] + x[2])
sage: a1 = matrix([[-7., -11.], [-11., 3.]])
sage: a2 = matrix([[7., -18.], [-18., 8.]])
sage: a3 = matrix([[-2., -8.], [-8., 1.]])
sage: a4 = matrix([[33., -9.], [-9., 26.]])
sage: b1 = matrix([[-21., -11., 0.], [-11., 10., 8.], [0.,   8., 5.]])
sage: b2 = matrix([[0.,  10.,  16.], [10., -10., -10.], [16., -10., 3.]])
```

```
sage: b3 = matrix([[-5.,   2., -17.], [2.,  -6.,   8.], [-17.,  8., 6.]])
sage: b4 = matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
sage: p.add_constraint(b1*x[0] + b2*x[1] + b3*x[2] <= b4)
sage: round(p.solve(),3)
-3.154
sage: round(p.get_backend().get_objective_value(),3)
-3.154
```

get_variable_value(*variable*)

Return the value of a variable given by the solver.

Note: Behaviour is undefined unless `solve` has been called before.

EXAMPLE:: sage: from sage.numerical.backends.generic_sdp_backend import get_solver sage: p = get_solver(solver = "cvxopt") sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False) sage: x = p.new_variable() sage: p.set_objective(x[0] - x[1] + x[2]) sage: a1 = matrix([[-7., -11.], [-11., 3.]]) sage: a2 = matrix([[7., -18.], [-18., 8.]]) sage: a3 = matrix([[-2., -8.], [-8., 1.]]) sage: a4 = matrix([[33., -9.], [-9., 26.]]) sage: b1 = matrix([[-21., -11., 0.], [-11., 10., 8.], [0., 8., 5.]]) sage: b2 = matrix([[0., 10., 16.], [10., -10., -10.], [16., -10., 3.]]) sage: b3 = matrix([[-5., 2., -17.], [2., -6., 8.], [-17., 8., 6.]]) sage: b4 = matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]]) sage: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4) sage: p.add_constraint(b1*x[0] + b2*x[1] + b3*x[2] <= b4) sage: round(p.solve(),3) -3.154 sage: round(p.get_backend().get_variable_value(0),3) -0.368 sage: round(p.get_backend().get_variable_value(1),3) 1.898 sage: round(p.get_backend().get_variable_value(2),3) -0.888

is_maximization()

Test whether the problem is a maximization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

ncols()

Return the number of columns/variables.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.nrows()
0
sage: p.add_variables(5)
4
sage: p.add_linear_constraints(2)
sage: p.nrows()
2
```

objective_coefficient (*variable*, *coeff=None*)

Set or get the coefficient of a variable in the objective function

INPUT:

- `variable` (integer) – the variable's id
- `coeff` (double) – its coefficient

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0.0
sage: p.objective_coefficient(0,2)
sage: p.objective_coefficient(0)
2.0
```

problem_name (*name='NULL'*)

Return or define the problem's name

INPUT:

- `name` (`char *`) – the problem's name. When set to `NULL` (default), the method returns the problem's name.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.problem_name("There once was a french fry")
sage: print p.problem_name()
There once was a french fry
```

row (*i*)

Return a row

INPUT:

- `index` (integer) – the constraint's id.

OUTPUT:

A pair `(indices, coeffs)` where `indices` lists the entries whose coefficient is nonzero, and to which `coeffs` associates their coefficient on the model of the `add_linear_constraint` method.

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
```

```
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(  [(0, matrix([[33., -9.], [-9., 26.]])) , (1,  matrix([[-7.,
sage: p.row(0)
([0, 1],
 [
[ 33.0000000000000 -9.00000000000000]
[-9.00000000000000  26.0000000000000],

[-7.00000000000000 -11.0000000000000]
[-11.0000000000000  3.00000000000000]
])
```

row_name (*index*)

Return the `index` th row name

INPUT:

- `index` (integer) – the row's id

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_linear_constraints(1, names="A")
sage: p.row_name(0)
'A'
```

set_objective (*coeff, d=0.0*)

Set the objective function.

INPUT:

- `coeff` – a list of real values, whose ith element is the coefficient of the ith variable in the objective function.
- `d` (double) – the constant term in the linear function (set to 0 by default)

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
sage: map(lambda x :p.objective_coefficient(x), range(5))
[1, 1, 2, 1, 3]
```

set_sense (*sense*)

Set the direction (maximization/minimization).

INPUT:

- `sense` (integer) :
  - +1 => Maximization
  - -1 => Minimization

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
```

```
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False
```

solve()
Solve the problem.

Note: This method raises SDPSolverException exceptions when the solution can not be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLE:

```
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1] + x[2])
sage: a1 = matrix([[-7., -11.], [-11., 3.]])
sage: a2 = matrix([[7., -18.], [-18., 8.]])
sage: a3 = matrix([[-2., -8.], [-8., 1.]])
sage: a4 = matrix([[33., -9.], [-9., 26.]])
sage: b1 = matrix([[-21., -11., 0.], [-11., 10., 8.], [0.,   8., 5.]])
sage: b2 = matrix([[0.,  10.,  16.], [10., -10., -10.], [16., -10., 3.]])
sage: b3 = matrix([[-5.,   2., -17.], [2.,  -6.,   8.], [-17.,  8., 6.]])
sage: b4 = matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]])
sage: p.add_constraint(a1*x[0] + a3*x[2] <= a4)
sage: p.add_constraint(b1*x[0] + b2*x[1] + b3*x[2] <= b4)
sage: round(p.solve(), 3)
-3.225
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1] + x[2])
sage: a1 = matrix([[-7., -11.], [-11., 3.]])
sage: a2 = matrix([[7., -18.], [-18., 8.]])
sage: a3 = matrix([[-2., -8.], [-8., 1.]])
sage: a4 = matrix([[33., -9.], [-9., 26.]])
sage: b1 = matrix([[-21., -11., 0.], [-11., 10., 8.], [0.,   8., 5.]])
sage: b2 = matrix([[0.,  10.,  16.], [10., -10., -10.], [16., -10., 3.]])
sage: b3 = matrix([[-5.,   2., -17.], [2.,  -6.,   8.], [-17.,  8., 6.]])
sage: b4 = matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
sage: p.add_constraint(b1*x[0] + b2*x[1] + b3*x[2] <= b4)
sage: round(p.solve(), 3)
-3.154
```

solver_parameter(*name*, *value=None*)
Return or define a solver parameter

INPUT:

- name (string) – the parameter
- value – the parameter's value if it is to be defined, or None (default) to obtain its current value.

Note: The list of available parameters is available at solver_parameter().

EXAMPLE:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver = "CVXOPT")
sage: p.solver_parameter("show_progress")
False
sage: p.solver_parameter("show_progress", True)
sage: p.solver_parameter("show_progress")
True
```

For more details on CVXOPT, see CVXOPT documentation.

CHAPTER

TWELVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[HPS08] J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer, 2008.

[ZBN97] C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software, Vol 23, Num. 4, pp.550–560, 1997.

PYTHON MODULE INDEX

n

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z