## CONTENTS

1 Knapsack Problems 1
2 Mixed Integer Linear Programming 7
3 Semidefinite Programming 39
4 Linear Functions and Constraints 53
5 Matrix/Vector-Valued Linear Functions: Parents 63
6 Matrix/Vector-Valued Linear Functions: Elements 67
7 Constraints on Linear Functions Tensored with a Free Module 69
8 Numerical Root Finding and Optimization 73
9 Interactive Simplex Method 83
10 Gauss-Legendre Integration for Vector-Valued Functions 139
11 Linear Optimization (LP) and Mixed Integer Linear Optimization (MIP) Solver backends 143
12 Semidefinite Optimization (SDP) Solver backends 265
13 Indices and Tables 279

Python Module Index 281

Index 283
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):

Maximize: $\sum_i b_i u_i$

Such that: $\sum_i b_i w_i \leq W$

$\forall i, b_i$ binary variable

(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]
```

```python
class sage.numerical.knapsack.Superincreasing(seq=None)
    Bases: SageObject
    A class for super-increasing sequences.

    Let \( L = (a_1, a_2, a_3, \ldots, a_n) \) 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()
```

(continues on next page)
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 $L = (a_1, a_2, a_3, \ldots, a_n)$ 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
```

**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
```

**subset_sum** (*N*)

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

Let \( S = (s_1, s_2, s_3, \ldots, s_n) \) be a non-empty sequence of non-negative integers, and let \( N \in \mathbb{Z} \) 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 [HPS2008].

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

```python
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]
```

sage.numerical.knapsack.knapsack(seq, binary, max=True, value_only=1, solver=False, verbose=None, integrality_tolerance=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 Mixed Integer Linear Programming (MILP) solver to be used. If set to None, the default one is used. For more information on MILP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
• verbose – integer (default: 0). Sets the level of verbosity. Set to 0 by default, which means quiet.
• integrality_tolerance – parameter for use with MILP solvers over an inexact base ring; see MixedIntegerLinearProgram.get_values().

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

```python
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:

```python
sage: from sage.numerical.knapsack import knapsack
sage: knapsack([1,1.5,0.5], max=2, value_only=True)
2.0
```
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 (Wikipedia article Optimization_(mathematics)) in the following form

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

with given $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^{m}$, $c \in \mathbb{R}^{n}$ 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 non-negative integer variables $w$ via $w = p.new_variable(integer=True, nonnegative=True)$.
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. 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`.

7. To check if everything is set up correctly, you can print the problem via `show`.

8. `Solve` it and print the solution.

The following example shows all these steps:

```python
sage: p = MixedIntegerLinearProgram(maximization=False, solver="GLPK")
sage: w = p.new_variable(integer=True, nonnegative=True)
sage: p.add_constraint(w[3] >= 1)
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=0.0, max=+oo)
  x_2 is an integer variable (min=0.0, max=+oo)
  x_3 is an integer variable (min=0.0, max=+oo)
sage: print('Objective Value: ' + format(p.solve()))
Objective Value: 2.0
sage: for i, v in sorted(p.get_values(w, convert=ZZ, tolerance=1e-3).items()):
    print(f'w_{i} = {v}')
  w_0 = 15
  w_1 = 10
  w_2 = 3
  w_3 = 2
```

Different backends compute with different base fields, for example:

```python
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 More about MIP variables

The underlying MILP backends always work with matrices where each column corresponds to a linear variable. The variable corresponding to the $i$-th column (counting from 0) is displayed as $x_i$.

MixedIntegerLinearProgram maintains a dynamic mapping from the arbitrary keys indexing the components of MIPVariable objects to the backend variables (indexed by nonnegative integers). Backend variables are created when a component of a MIPVariable is accessed.

To make your code more readable, you can construct one or several MIPVariable objects that can be arbitrarily named and indexed. This can be done by calling new_variable() several times, or by the following special syntax:

```
sage: mip.<a,b> = MixedIntegerLinearProgram(solver='GLPK')
sage: a
MIPVariable a with 0 real components
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
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)
```

Upper/lower bounds on a variable can be specified either as separate constraints (see add_constraint) or using the methods set_max and set_min respectively.

2.4 The default MIP variable

As a special shortcut, it is not necessary to call new_variable(). A MixedIntegerLinearProgram has a default MIPVariable, whose components are obtained by using the syntax mip[key], where key is an arbitrary key:

```
sage: mip = MixedIntegerLinearProgram(solver='GLPK')
5 + x_0 + 2*x_1
```
2.5 Index of functions and methods

Below are listed the methods of \texttt{MixedIntegerLinearProgram}. This module also implements the \texttt{MIPSolverException} exception, as well as the \texttt{MIPVariable} class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{add_constraint()}</td>
<td>Adds a constraint to the \texttt{MixedIntegerLinearProgram}</td>
</tr>
<tr>
<td>\texttt{base_ring()}</td>
<td>Return the base ring</td>
</tr>
<tr>
<td>\texttt{best_known_objective_bound()}</td>
<td>Return the value of the currently best known bound</td>
</tr>
<tr>
<td>\texttt{constraints()}</td>
<td>Returns a list of constraints, as 3-tuples</td>
</tr>
<tr>
<td>\texttt{default_variable()}</td>
<td>Return the default \texttt{MIPVariable} of \texttt{self}</td>
</tr>
<tr>
<td>\texttt{get_backend()}</td>
<td>Returns the backend instance used</td>
</tr>
<tr>
<td>\texttt{get_max()}</td>
<td>Returns the maximum value of a variable</td>
</tr>
<tr>
<td>\texttt{get_min()}</td>
<td>Returns the minimum value of a variable</td>
</tr>
<tr>
<td>\texttt{get_objective_value()}</td>
<td>Return the value of the objective function</td>
</tr>
<tr>
<td>\texttt{get_relative_objective_gap()}</td>
<td>Return the relative objective gap of the best known solution</td>
</tr>
<tr>
<td>\texttt{get_values()}</td>
<td>Return values found by the previous call to \texttt{solve()}</td>
</tr>
<tr>
<td>\texttt{is_binary()}</td>
<td>Tests whether the variable ( e ) is binary</td>
</tr>
<tr>
<td>\texttt{is_integer()}</td>
<td>Tests whether the variable is an integer</td>
</tr>
<tr>
<td>\texttt{is_real()}</td>
<td>Tests whether the variable is real</td>
</tr>
<tr>
<td>\texttt{linear_constraints_parent()}</td>
<td>Return the parent for all linear constraints</td>
</tr>
<tr>
<td>\texttt{linear_functions_parent()}</td>
<td>Return the parent for all linear functions</td>
</tr>
<tr>
<td>\texttt{new_variable()}</td>
<td>Returns an instance of \texttt{MIPVariable} associated</td>
</tr>
<tr>
<td>\texttt{number_of_constraints()}</td>
<td>Returns the number of constraints assigned so far</td>
</tr>
<tr>
<td>\texttt{number_of_variables()}</td>
<td>Returns the number of variables used so far</td>
</tr>
<tr>
<td>\texttt{polyhedron()}</td>
<td>Returns the polyhedron defined by the Linear Program</td>
</tr>
<tr>
<td>\texttt{remove_constraint()}</td>
<td>Removes a constraint from \texttt{self}</td>
</tr>
<tr>
<td>\texttt{remove_constraints()}</td>
<td>Remove several constraints</td>
</tr>
<tr>
<td>\texttt{set_binary()}</td>
<td>Sets a variable or a \texttt{MIPVariable} as binary</td>
</tr>
<tr>
<td>\texttt{set_integer()}</td>
<td>Sets a variable or a \texttt{MIPVariable} as integer</td>
</tr>
<tr>
<td>\texttt{set_max()}</td>
<td>Sets the maximum value of a variable</td>
</tr>
<tr>
<td>\texttt{set_min()}</td>
<td>Sets the minimum value of a variable</td>
</tr>
<tr>
<td>\texttt{set_objective()}</td>
<td>Sets the objective of the \texttt{MixedIntegerLinearProgram}</td>
</tr>
<tr>
<td>\texttt{set_problem_name()}</td>
<td>Sets the name of the \texttt{MixedIntegerLinearProgram}</td>
</tr>
<tr>
<td>\texttt{set_real()}</td>
<td>Sets a variable or a \texttt{MIPVariable} as real</td>
</tr>
<tr>
<td>\texttt{show()}</td>
<td>Displays the \texttt{MixedIntegerLinearProgram} in a human-readable</td>
</tr>
<tr>
<td>\texttt{solve()}</td>
<td>Solves the \texttt{MixedIntegerLinearProgram}</td>
</tr>
<tr>
<td>\texttt{solver_parameter()}</td>
<td>Return or define a solver parameter</td>
</tr>
<tr>
<td>\texttt{sum()}</td>
<td>Efficiently computes the sum of a sequence of \texttt{LinearFunction} elements</td>
</tr>
<tr>
<td>\texttt{write_lp()}</td>
<td>Write the linear program as a LP file</td>
</tr>
<tr>
<td>\texttt{write_mps()}</td>
<td>Write the linear program as a MPS file</td>
</tr>
</tbody>
</table>

AUTHORS:

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

\texttt{exception} \texttt{sage.numerical.mip.MIPSolverException}

Bases: \texttt{RuntimeError}
Exception raised when the solver fails.

EXAMPLES:

```python
define: from sage.numerical.mip import MIPSolverException
define: e = MIPSolverException("Error")
define: e
MIPSolverException('Error')
define: print(e)
Error
```

class sage.numerical.mip.MIPVariable

Bases: FiniteFamily

MIPVariable is a variable used by the class MixedIntegerLinearProgram.

Warning: You should not instantiate this class directly. Instead, use MixedIntegerLinearProgram.

new_variable().

copy_for_mip (mip)

Returns a copy of self suitable for a new MixedIntegerLinearProgram instance mip.

For this to make sense, mip should have been obtained as a copy of self.mip().

EXAMPLES:

```python
define: p = MixedIntegerLinearProgram(solver='GLPK')
define: pv = p.new_variable(nonnegative=True)
define: pv[0]
x_0
define: q = copy(p)
define: qv = pv.copy_for_mip(q)
define: pv[77]
x_1
define: q.number_of_variables()
2
define: qv.number_of_variables()
1
define: qv[33]
x_1
define: p.number_of_variables()
2
define: qv.number_of_variables()
2
define: p = MixedIntegerLinearProgram(solver='GLPK')
define: pv = p.new_variable(indices=[3, 7])
define: q = copy(p)
define: qv = pv.copy_for_mip(q)
define: qv[3]
x_0
define: qv[5]
Traceback (most recent call last):
... 
IndexError: 5 does not index a component of MIPVariable with 2 real components
```

items()
Return the pairs (keys, value) contained in the dictionary.

EXAMPLES:

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

`keys()`

Return the keys already defined in the dictionary.

EXAMPLES:

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

`mip()`

Returns the `MixedIntegerLinearProgram` in which `self` is a variable.

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=True)
sage: p == v.mip()
True
```

`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:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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:
```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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()**

Return the symbolic variables associated to the current dictionary.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[0] + v[1])
sage: sorted(v.values(), key=str)
[x_0, x_1]
```

---

**class** `sage.numerical.mip.MixedIntegerLinearProgram`

**Bases:** `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 thematic tutorial on Linear Programming (Mixed Integer) or Wikipedia article Linear_programming for further information on linear programming, and the MILP module for its use in Sage.

**INPUT:**

- **solver** – selects a solver; see `Solvers (backends)` for more information and installation instructions for optional solvers.
  - `solver="GLPK"`: The GNU Linear Programming Kit.
  - `solver="GLPK/exact"`: GLPK’s implementation of an exact rational simplex method.
  - `solver="Coin"`: The COIN-OR CBC (COIN Branch and Cut) solver.
  - `solver="CPLEX"`, provided by the proprietary IBM ILOG CPLEX Optimization Studio.
  - `solver="Gurobi"`: The proprietary Gurobi solver.
  - `solver="CVXOPT"`: See the CVXOPT web site.
  - `solver="PPL"`: An exact rational solver (for small scale instances) provided by the Parma Polyhedra Library (PPL).
  - `solver="InteractiveLP"`: A didactical implementation of the revised simplex method in Sage. It works over any exact ordered field, the default is QQ.
  - If `solver=None` (default), the default solver is used (see `default_mip_solver()`).
  - `solver` can also be a callable (such as a class), see `sage.numerical.backends.generic_backend.get_solver()` for examples.

---

2.5. Index of functions and methods 13
• 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.

See also:

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

EXAMPLES:

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

```python
sage: # needs sage.graphs
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization=True, solver='GLPK')
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(sort=False, labels=None):
  ....:   p.add_constraint(b[u] + b[v], max=1)
sage: p.solve(objective_only=True)
sage: 4.0
```

`add_constraint(linear_function, max=None, min=None, name=None, return_indices=False)`

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 \( A \leq B, A \geq B, A \leq B \leq C, A \geq B \geq C \) 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.

• `return_indices` – boolean (optional, default False), whether to return the indices of the added constraints.
OUTPUT:
The row indices of the constraints added, if return_indices is true and the backend guarantees that removing them again yields the original MIP, None otherwise.

To set a lower and/or upper bound on the variables use the methods set_min and/or set_max of Mixed-IntegerLinearProgram.

EXAMPLES:
Consider the following linear program:

Maximize:
   \( x + 5 \cdot y \)
Constraints:
   \( x + 0.2 \cdot y \leq 4 \)
   \( 1.5 \cdot x + 3 \cdot y \leq 4 \)
Variables:
   \( x \) is Real (min = 0, max = None)
   \( y \) is Real (min = 0, max = None)

It can be solved as follows:

```python
sage: p = MixedIntegerLinearProgram(maximization=True, solver='GLPK')
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 \( x[5] + 3 \cdot x[7] \leq x[6] + 3 \) to a MixedIntegerLinearProgram.

The first one consists in giving add_constraint this very expression:

```python
```

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

```python
```

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

```python
```

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

```python
sage: p = MixedIntegerLinearProgram(maximization=True, solver='GLPK')
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 also be combined into a single vector-valued constraint:
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, solver='GLPK')
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  3.000000000000000]
sage: p = MixedIntegerLinearProgram(maximization=True, solver='GLPK')
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
```

**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
sage: from sage.rings.qqbar import AA
sage: p = MixedIntegerLinearProgram(solver='InteractiveLP', base_ring=AA)
sage: p.base_ring()
Algebraic Real Field
sage: # needs sage.groups sage.rings.number_field
d = polytopes.dodecahedron()
```
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.

EXAMPLES:

```
sage: # needs sage.graphs
go = 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 go))
sage: for v in go:
    ....: p.add_constraint(b[v] + p.sum(b[u] for u in go.neighbors(v)) <= 1)
sage: p.add_constraint(b[v] == 1)  # Force an easy non-0 solution
sage: p.solve()  # rel toi 100
1.0
sage: p.best_known_objective_bound()  # random
48.0
```

constraints (indices=\texttt{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.

EXAMPLES:

First, let us define a small LP:
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)

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)]

default_variable()

Return the default MIPVariable of self.

EXAMPLES:

sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: p.default_variable()
MIPVariable with 0 real components

get_backend()

Returns the backend instance used.

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

EXAMPLES:

This example uses the simplex algorithm 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: ans = p.solve()
GLPK Simplex Optimizer...
2 rows, 2 columns, 4 non-zeros
* 0: obj =  7.000000000e+00 inf =  0.000e+00 (2)
* 2: obj =  9.400000000e+00 inf =  0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
sage: ans  # rel tol 1e-5
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.

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_max(v[1])
6.0
```

\textbf{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.

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=True)
sage: p.set_objective(v[1])
sage: p.get_min(v[1])
0.0
```

\textbf{get\_objective\_value}\()\)

Return the value of the objective function.

\textbf{Note:} Behaviour is undefined unless \texttt{solve} has been called before.

EXAMPLES:

```python
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
```

\textbf{get\_relative\_objective\_gap}\()\)

Return the relative objective gap of the best known solution.
For a minimization problem, this value is computed by 
\[
\frac{\text{bestinteger} - \text{bestobjective}}{10 + |\text{bestobjective}|},
\]
where \text{bestinteger} is the value returned by \textit{get_objective_value()} and \text{bestobjective} is the value returned by \textit{best_known_objective_bound()}. For a maximization problem, the value is computed by 
\[
\frac{\text{bestobjective} - \text{bestinteger}}{10 + |\text{bestobjective}|}.
\]

\textbf{Note:} Has no meaning unless \textit{solve} has been called before.

**EXAMPLES:**

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

\textit{get_values (convert=None, tolerance=None, *lists)}

Return values found by the previous call to \textit{solve()}. 

**INPUT:**

- \*\textit{lists} – any instance of \texttt{MIPVariable} (or one of its elements), or lists of them.
- \texttt{convert} – \texttt{None} (default), \texttt{ZZ}, \texttt{bool}, or \texttt{True}.
  - if \texttt{convert}=	exttt{None} (default), return all variable values as the backend provides them, i.e., as an element of \texttt{base_ring()} or a float.
  - if \texttt{convert}=	exttt{ZZ}, convert all variable values from the \texttt{base_ring()} by rounding to the nearest integer.
  - if \texttt{convert}=	exttt{bool}, convert all variable values from the \texttt{base_ring()} by rounding to 0/1 and converting to bool.
  - if \texttt{convert}=	exttt{True}, use \texttt{ZZ} for MIP variables declared integer or binary, and convert the values of all other variables to the \texttt{base_ring()}.
- \texttt{tolerance} – \texttt{None}, a positive real number, or \texttt{0} (if \texttt{base_ring()} is an exact ring). Required if \texttt{convert} is not \texttt{None} and any integer conversion is to be done. If the variable value differs from the nearest integer by more than \texttt{tolerance}, raise a \texttt{RuntimeError}.

**OUTPUT:**

- Each instance of \texttt{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 \texttt{MIPVariable} is replaced by its corresponding numerical value.

\textbf{Note:} While a variable may be declared as binary or integer, its value is an element of the \texttt{base_ring()}, or for the numerical solvers, a float.
For the numerical solvers, `base_ring()` is RDF, an inexact ring. Code using `get_values` should always account for possible numerical errors.

Even for variables declared as binary or integer, or known to be an integer because of the mathematical properties of the model, the returned values cannot be expected to be exact integers. This is normal behavior of the numerical solvers.

For correct operation, any user code needs to avoid exact comparisons (==, !=) and instead allow for numerical tolerances. The magnitude of the numerical tolerances depends on both the model and the solver.

The arguments `convert` and `tolerance` facilitate writing correct code. See examples below.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x = p.new_variable(nonnegative=True)
sage: y = p.new_variable(nonnegative=True)
sage: p.solve()

6.0

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

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

2.0

```python
sage: type(_)
<class 'float'>
```

To convert the value to the `base_ring()`:

```python
sage: p.get_values(y[2,9], convert=True)

2.0

sage: _.parent()

Real Double Field
```

To get a dictionary identical to x containing the values for the corresponding variables in the optimal solution:

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

[3, 5]
```

Obviously, it also works with variables of higher dimension:

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

We could also have tried

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

Or:

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

Using `convert` and `tolerance`. First, a binary knapsack:
sage: p = MixedIntegerLinearProgram(solver='GLPK')
\sage: x = p.new_variable(binary=True)
\sage: p.set_objective(3*x[1] + 4*x[2] + 5*x[3])
\sage: p.solve()
8.0
\sage: x_opt = p.get_values(x); x_opt
\{1: 1.0, 2: 0.0, 3: 1.0\}
\sage: type(x_opt[1])
<class 'float'>
\sage: x_opt_ZZ = p.get_values(x, convert=True, tolerance=1e-6); x_opt_ZZ
\{1: 1, 2: 0, 3: 1\}
\sage: x_opt_ZZ[1].parent()
Integer Ring
\sage: x_opt_bool = p.get_values(x, convert=bool, tolerance=1e-6); x_opt_bool
\{ab: 0, at: 1, ba: 1, bt: 0, sa: 0, sb: 1\}

Thanks to total unimodularity, single-commodity network flow problems with integer capacities and integer supplies/demands have integer vertex solutions. Hence the integrality of solutions is mathematically guaranteed in an optimal solution if we use the simplex algorithm. A numerical LP solver based on the simplex method such as GLPK will return an integer solution only up to a numerical error. Hence, for correct operation, we should use \texttt{tolerance}:

\textbf{interactive\_lp\_problem\ (form='standard')}

Returns an InteractiveLPProblem and, if available, a basis.

INPUT:

- \texttt{form} – (default: "standard") a string specifying return type: either \texttt{None}, or \texttt{"std"} or \texttt{"standard"}, respectively returns an instance of InteractiveLPProblem or of InteractiveLPProblemStandardForm

OUTPUT:

A 2-tuple consists of an instance of class InteractiveLPProblem or InteractiveLPProblemStandardForm that is constructed based on a given MixedIntegerLinearProgram, and a list of basic variables (the basis) if standard form is chosen (by default), otherwise \texttt{None}.

All variables must have 0 as lower bound and no upper bound.

EXAMPLES:

\texttt{sage: p = MixedIntegerLinearProgram(names=['m'], solver='GLPK')}
\texttt{sage: x = p.new_variable(nonnegative=True)}
sage: y = p.new_variable(nonnegative=True, name='n')

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

sage: p.add_constraint(x[0] + x[1] - 7*y[0] + v[0] <= 2, name='K')

sage: p.add_constraint(x[1] + 2*y[0] - v[0] <= 3)

sage: p.add_constraint(5*x[0] + y[0] <= 21, name='L')

sage: p.set_objective(2*x[0] + 3*x[1] + 4*y[0] + 5*v[0])

sage: lp, basis = p.interactive_lp_problem()

sage: basis
['K', 'w_1', 'L']

sage: lp.constraint_coefficients()

\begin{bmatrix}
1.0 & 1.0 & -7.0 & 1.0 \\
0.0 & 1.0 & 2.0 & -1.0 \\
5.0 & 0.0 & 1.0 & 0.0
\end{bmatrix}

sage: lp.b()
(2.0, 3.0, 21.0)

sage: lp.objective_coefficients()
(2.0, 3.0, 4.0, 5.0)

sage: lp.decision_variables()
(m_0, m_1, n_0, x_3)

sage: view(lp)  #not tested

sage: d = lp.dictionary(*basis)

sage: view(d)  #not tested

\textbf{is\_binary}(e)

Tests whether the variable \( e \) is binary. Variables are real by default.

\textbf{INPUT:}

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

\textbf{OUTPUT:}

True if the variable \( e \) is binary; False otherwise.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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
\end{verbatim}

\textbf{is\_integer}(e)

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

\textbf{INPUT:}

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

\textbf{OUTPUT:}

True if the variable \( e \) is an integer; False otherwise.

\textbf{EXAMPLES:}
\begin{verbatim}
  sage: p = MixedIntegerLinearProgram(solver='GLPK')
  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.

EXAMPLES:
\begin{verbatim}
  sage: p = MixedIntegerLinearProgram(solver='GLPK')
  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
\end{verbatim}

linear_constraints_parent()
Return the parent for all linear constraints

See linear_functions for more details.

EXAMPLES:
\begin{verbatim}
  sage: p = MixedIntegerLinearProgram(solver='GLPK')
  sage: p.linear_constraints_parent()
  Linear constraints over Real Double Field
\end{verbatim}

linear_functions_parent()
Return the parent for all linear functions

EXAMPLES:
\begin{verbatim}
  sage: p = MixedIntegerLinearProgram(solver='GLPK')
  sage: p.linear_functions_parent()
  Linear functions over Real Double Field
\end{verbatim}

new_variable(real=False, binary=False, integer=False, nonnegative=False, name='', indices=None)
Return a new MIPVariable instance.

A new variable \( x \) is defined by:
\end{verbatim}
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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.
- `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.
- `indices` — (optional) an iterable of keys; components corresponding to these keys are created in order, and access to components with other keys will raise an error; otherwise components of this variable can be indexed by arbitrary keys and are created dynamically on access

OUTPUT:

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

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x = p.new_variable(); x
MIPVariable with 0 real components
sage: x0 = x[0]; x0
x_0

By default, variables are unbounded:

```python
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

```python
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(solver='GLPK')
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(solver='GLPK')
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()  
Return the number of constraints assigned so far.

EXAMPLES:

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: 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 $m \leq c^T x \leq M$, with $m < M$, into equations, by adding extra variables: $c^T x + y = M$, $0 \leq y \leq M - m$.

EXAMPLES:
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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 \texttt{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 \(\leq\) \texttt{linear_function} \(\leq\) 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 \texttt{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(solver='GLPK')
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 RDF^2 defined as the convex hull of 4 vertices

3-D Polyhedron:

sage: p = MixedIntegerLinearProgram(solver='GLPK')
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 RDF^3 defined as the convex hull of 8 vertices
An empty polyhedron:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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 RDF^3
```

An unbounded polyhedron:

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

A square (see github issue #14395)

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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 RDF^2 defined as the convex hull of 4 vertices
```

We can also use a backend that supports exact arithmetic:

```python
sage: p = MixedIntegerLinearProgram(solver='PPL')
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.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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()
sage: p.remove_constraint(0)
```

Maximization:

Constraints:

\[ x_0 + x_1 \leq 10.0 \]

(continues on next page)
remove_constraints(constraints)

Remove several constraints.

INPUT:

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

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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
  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:

```python
sage: p = MixedIntegerLinearProgram(check_redundant=True, solver='GLPK')
sage: x, y = p[0], p[1]
sage: p.add_constraint(x + y, max=10)
sage: for i in range(10):
    ....:     p.add_constraint(x - y, max=10)
sage: p.add_constraint(x, max=4)
```

(continues on next page)
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 range(10):
....:   p.add_constraint(x - y, max=10)

set_binary(ee)
Sets a variable or a MIPVariable as binary.

INPUT:

• ee – An instance of MIPVariable or one of its elements.

EXAMPLES:

sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x = p.new_variable(nonnegative=True)

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

sage: p.set_binary(x)

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.

EXAMPLES:

sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: x = p.new_variable(nonnegative=True)

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

sage: p.set_integer(x)

It is still possible, though, to set one of these variables as binary while keeping the others as they are:
**set_max** \((v, \text{max})\)

Sets the maximum value of a variable.

**INPUT:**
- \(v\) – a variable.
- \(\text{max}\) – the maximum value the variable can take. When \(\text{max}=None\), the variable has no upper bound.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=\text{True})
sage: p.set_objective(v[1])
sage: p.set_max(v[1])
sage: p.get_max(v[1])
6.0
```

With a **MIPVariable** as an argument:

```python
sage: vv = p.new_variable(real=\text{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
```  

**set_min** \((v, \text{min})\)

Sets the minimum value of a variable.

**INPUT:**
- \(v\) – a variable.
- \(\text{min}\) – the minimum value the variable can take. When \(\text{min}=None\), the variable has no lower bound.

**See also:**
- **get_min()** – get the minimum value of a variable.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver='GLPK')
sage: v = p.new_variable(nonnegative=\text{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
```  

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)
5.0
sage: p.get_min(vv[0])
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 or any number when just looking for a feasible solution)

**EXAMPLES:**

Let’s solve the following linear program:

Maximize:

\[ x + 5 \times y \]

Constraints:

\[ x + 0.2 \times y \leq 4 \]
\[ 1.5 \times x + 3 \times y \leq 4 \]

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, solver='GLPK')
x = p.new_variable(nonnegative=True)
p.set_objective(x[1] + 5*x[2])
p.add_constraint(x[1] + 2/10*x[2], max=4)
p.add_constraint(1.5*x[1] + 3*x[2], max=4)
round(p.solve(),5)
6.66667
```

```sage
p.set_objective(None)
_ = p.solve()
```

**set_problem_name**(name)

Sets the name of the MixedIntegerLinearProgram.

**INPUT:**

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

**EXAMPLES:**

```sage
p = MixedIntegerLinearProgram(solver='GLPK')
p.set_problem_name("Test program")
p
```

**set_real**(ee)

Sets a variable or a MIPVariable as real.
INPUT:

• ee – An instance of MIPVariable or one of its elements.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(solver='GLPK')
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:
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)
```

(continues on next page)
**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, no additional assumption is made on the domain of an LP variable. See *set_min()* and *set_max()* to change it.

**EXAMPLES:**

Consider the following linear program:

Maximize:

\[ x + 5 \times y \]

Constraints:

\[ x + 0.2 \times y \leq 4 \]
\[ 1.5 \times x + 3 \times y \leq 4 \]

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, solver='GLPK')
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: # needs sage.graphs
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization=True, solver='GLPK')
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(sort=False, 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(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.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, SCIP, 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:
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")
  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](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.

**EXAMPLES:**

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

```
 50.0
```

**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
 sage: p = MixedIntegerLinearProgram(solver='GLPK')
 sage: v = p.new_variable(nonnegative=True)
```
The following command:

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

is much more efficient than:

```sage
s = sum(v[i] for i in range(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.

**EXAMPLES:**

```sage
p = MixedIntegerLinearProgram(solver="GLPK")
x = p.new_variable(nonnegative=True)
p.set_objective(x[1] + x[2])
p.add_constraint(-3*x[1] + 2*x[2], max=2)
import tempfile
with tempfile.NamedTemporaryFile(suffix=".lp") as f:
    p.write_lp(f.name)
```

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

**EXAMPLES:**

```sage
p = MixedIntegerLinearProgram(solver="GLPK")
x = p.new_variable(nonnegative=True)
p.set_objective(x[1] + x[2])
p.add_constraint(-3*x[1] + 2*x[2], max=2, name="OneConstraint")
import tempfile
with tempfile.NamedTemporaryFile(suffix=".mps") as f:
    p.write_mps(f.name)
```

For information about the MPS file format, see [Wikipedia article MPS_(format)](https://en.wikipedia.org/wiki/MPS_%28format%29)
A semidefinite program (SDP) is an optimization problem (Wikipedia article Optimization_(mathematics)) of the following form

$$\min \sum_{i,j=1}^{n} C_{ij} X_{ij} \quad \text{(Dual problem)}$$

Subject to:

$$\sum_{i,j=1}^{n} A_{ijk} X_{ij} = b_k, \quad k = 1 \ldots m$$
$$X \succeq 0$$

where the $X_{ij}$, $1 \leq i, j \leq n$ are $n^2$ variables satisfying the symmetry conditions $x_{ij} = x_{ji}$ for all $i, j$, the $C_{ij} = C_{ji}$, $A_{ijk} = A_{kji}$ and $b_k$ are real coefficients, and $X$ is positive semidefinite, i.e., all the eigenvalues of $X$ are nonnegative.

The closely related dual problem of this one is the following, where we denote by $A_k$ the matrix $(A_{kij})$ and by $C$ the matrix $(C_{ij})$,

$$\max \sum_{k} b_k x_k \quad \text{(Primal problem)}$$

Subject to:

$$\sum_{k} x_k A_k \preceq C.$$

Here $(x_1, \ldots, x_m)$ is a vector of scalar variables. A wide variety of problems in optimization can be formulated in one of these two standard forms. Then, solvers are able to calculate an approximation to a solution. Here we refer to the latter problem as primal, and to the former problem as dual. The optimal value of the dual is always at least the optimal value of the primal, and usually (although not always) they are equal.

For instance, suppose you want to maximize $x_1 - x_0$ subject to

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

An SDP can give you an answer to the problem above. Here is how it's done:

1. You have to create an instance of `SemidefiniteProgram`.
2. Create a dictionary $x$ of integer variables via `new_variable()`, for example doing $x = p.new_variable()` if $p$ is the name of the SDP instance.
3. Add those two matrix inequalities as inequality constraints via `add_constraint()`.
4. Add another matrix inequality to specify nonnegativity of $x$.
5. Specify the objective function via `set_objective()`. In our case it is $x_1 - x_0$. If it is a pure constraint satisfaction problem, specify it as `None`.
6. To check if everything is set up correctly, you can print the problem via `show`.


7. Solve it and print the solution.

The following example shows all these steps:

```python
sage: p = SemidefiniteProgram()
sage: x = p.new_variable()
sage: p.set_objective(x[1] - x[0])
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: c1 = matrix([[1.0, 0], [0,0]], sparse=True)
sage: c2 = matrix([[0.0, 0], [0,1]], sparse=True)
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(c1*x[0] + c2*x[1] >= matrix.zero(2,2, sparse=True))
sage: # needs cvxopt
sage: p.solver_parameter("show_progress", True)
sage: opt = p.solve()
```

Optimal solution found.

```python
sage: print(Objective Value: {}) format(N(opt,3))  
1.0
sage: [N(x, 3) for x in sorted(p.get_values(x).values())]
[3.0e-8, 1.0]
sage: p.show()
Maximization:
  x_0 - x_1
Constraints:
  constraint_0: [3.0 4.0]x_0 + [1.0 2.0]x_1 <= [5.0 6.0]  
  constraint_1: [2.0 2.0]x_0 + [1.0 1.0]x_1 <= [3.0 3.0]  
  constraint_2: [ 0.0 0.0]x_0 + [-1.0 0.0]x_1 <= [0 0]
Variables:
  x_0, x_1
```

Most solvers, e.g. the default Sage SDP solver CVXOPT, solve simultaneously the pair of primal and dual problems. Thus we can get the optimizer \(\mathbf{X}\) of the dual problem as follows, as diagonal blocks, one per each constraint, via `dual_variable()`. E.g.:

```python
sage: p.dual_variable(1)  
[ 85555.0 -85555.0]  
[-85555.0 85555.0]
```

We can see that the optimal value of the dual is equal (up to numerical noise) to `opt`:

```python
sage: opt - ((p.dual_variable(0)*a3).trace() + (p.dual_variable(1)*b3).trace())
0.0
```

Dual variable blocks at optimality are orthogonal to “slack variables”, that is, matrices \( \mathbf{C} - \sum_{\mathbf{A}_k} \mathbf{X}_k \mathbf{A}_k \), cf. (Primal problem) above, available via `slack()`. E.g.:
More interesting example, the Lovasz theta of the 7-gon:

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

Unlike in the previous example, the slack variable is very far from 0:

```
sage: p.slack(0).trace()  # tol 1e-14
1.0
```

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

```
sage: # needs cvxopt
c = SemidefiniteProgram(solver='cvxopt')
x = c.new_variable()
0.5 + 3/2*x[1]
```

For representing an SDP with exact data, there is another backend:

```
sage: from sage.numerical.backends.matrix_sdp_backend import MatrixSDPBackend
c = SemidefiniteProgram(solver=MatrixSDPBackend, base_ring=QQ)
x = c.new_variable()
1/2 + 3/2 * x[1]
```

### 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()
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:
Numerical Optimization, Release 10.3

sage: a[4, 'string', QQ]
x_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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_constraint()</td>
<td>Adds a constraint to the <em>SemidefiniteProgram</em></td>
</tr>
<tr>
<td>base_ring()</td>
<td>Return the base ring</td>
</tr>
<tr>
<td>dual_variable()</td>
<td>Return optimal dual variable block</td>
</tr>
<tr>
<td>get_backend()</td>
<td>Return the backend instance used</td>
</tr>
<tr>
<td>get_values()</td>
<td>Return values found by the previous call to solve()</td>
</tr>
<tr>
<td>linear_constraints_parent()</td>
<td>Return the parent for all linear constraints</td>
</tr>
<tr>
<td>linear_function()</td>
<td>Construct a new linear function</td>
</tr>
<tr>
<td>linear_functions_parent()</td>
<td>Return the parent for all linear functions</td>
</tr>
<tr>
<td>new_variable()</td>
<td>Return an instance of <em>SDPVariable</em> associated to the <em>SemidefiniteProgram</em></td>
</tr>
<tr>
<td>number_of_constraints()</td>
<td>Return the number of constraints assigned so far</td>
</tr>
<tr>
<td>number_of_variables()</td>
<td>Return the number of variables used so far</td>
</tr>
<tr>
<td>set_objective()</td>
<td>Set the objective of the <em>SemidefiniteProgram</em></td>
</tr>
<tr>
<td>set_problem_name()</td>
<td>Set the name of the <em>SemidefiniteProgram</em></td>
</tr>
<tr>
<td>slack()</td>
<td>Return the slack variable block at the optimum</td>
</tr>
<tr>
<td>show()</td>
<td>Display the <em>SemidefiniteProgram</em> in a human-readable way</td>
</tr>
<tr>
<td>solve()</td>
<td>Solve the <em>SemidefiniteProgram</em></td>
</tr>
<tr>
<td>solver_parameter()</td>
<td>Return or define a solver parameter</td>
</tr>
<tr>
<td>sum()</td>
<td>Efficiently compute the sum of a sequence of <em>LinearFunction</em> elements</td>
</tr>
</tbody>
</table>

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

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

Bases: *RuntimeError*

Exception raised when the solver fails.

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

**EXAMPLES:**
class sage.numerical.sdp.SDPVariable

Bases: Element

SDPVariable is a variable used by the class SemidefiniteProgram.

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

items()

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

EXAMPLES:

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

keys()

Return the keys already defined in the dictionary.

EXAMPLES:

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

values()

Return the symbolic variables associated to the current dictionary.

EXAMPLES:

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

class sage.numerical.sdp.SDPVariableParent

Bases: 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.

3.2. Index of functions and methods
**Element**

alias of *SDPVariable*

class *sage.numerical.sdp.SemidefiniteProgram*

**Bases:** *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 website.
  - 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:

```python
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: N(p.solve(), 2)  # needs cvxopt
-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 <= C, A >= B >= C or A == B. In this case, arguments min and max will be ignored.
• name – A name for the constraint.

EXAMPLES:
Let’s solve the following semidefinite program:

\[
\begin{align*}
\text{maximize} & \quad x + 5y \\
\text{subject to} & \quad \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} x + \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} y \preceq \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}
\end{align*}
\]

This SDP can be solved as follows:

```python
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: N(p.solve(), digits=3) # needs cvxopt
16.2
```

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

```python
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: N(p.solve(), digits=3) # needs cvxopt
16.2
```

**base_ring()**

Return the base ring.

**OUTPUT:**

A ring. The coefficients that the chosen solver supports.

**EXAMPLES:**

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

**dual_variable(i, sparse=False)**

The i-th dual variable.

Available after self.solve() is called, otherwise the result is undefined.

**INPUT:**

• index (integer) – the constraint’s id

**OUTPUT:**

The matrix of the i-th dual variable.
EXAMPLES:

Dual objective value is the same as the primal one:

```
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.solve()  # tol 1e-08
-3.0
```

```
sage: x = p.get_values(x).values()
sage: -(a3*p.dual_variable(0)).trace() - (b3*p.dual_variable(1)).trace()  # tol 1e-07
-3.0
```

Dual variable is orthogonal to the slack

```
sage: # needs cvxopt
sage: p.solve()  # tol 1e-08
-3.0
sage: x = p.get_values(x).values()
sage: -(a3*p.dual_variable(0)).trace() - (b3*p.dual_variable(1)).trace()  # tol 1e-07
-3.0
```

```
sage: # needs cvxopt
sage: g = sum((p.slack(j)*p.dual_variable(j)).trace() for j in range(2)); g  # tol 1.2e-08
0.0
```

gen(i)

Return the linear variable \( x_i \).

EXAMPLES:

```
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()

Return the backend instance used.

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

EXAMPLES:

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]
```

(continues on next page)
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.

**EXAMPLES:**

```python
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: N(p.solve(), 3)  # needs cvxopt
-3.0
```

To return the optimal value of x[3]:

```python
sage: N(p.get_values(x[3]), 3)  # needs cvxopt
-1.0
```

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

```python
sage: x_sol = p.get_values(x)  # needs cvxopt
sage: sorted(x_sol)  # needs cvxopt
[3, 5]
```

**linear_constraints_parent()**

Return the parent for all linear constraints.

See linear_functions for more details.

**EXAMPLES:**

```python
```
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:

- \( \text{dim} \) – integer. Defines the dimension of the dictionary. If \( x \) has dimension 2, its fields will be of the form \( x[\text{key1}][\text{key2}] \). Deprecated.

- \( \text{name} \) – string. Associates a name to the variable.

EXAMPLES:

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()

Return the number of constraints assigned so far.

EXAMPLES:
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()

Return the number of variables used so far.

EXAMPLES:

sage: p = SemidefiniteProgram()
sage: a = matrix([[1, 2.], [2., 3.]])
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)

EXAMPLES:

Let’s solve the following semidefinite program:

\[
\begin{align*}
\text{maximize} & \quad x + 5y \\
\text{subject to} & \quad \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} x + \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} y \preceq \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}
\end{align*}
\]

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: N(p.solve(), digits=3) #...

← needs cvxopt
16.2
sage: p.set_objective(None)
sage: _ = p.solve() #...

← needs cvxopt

3.2. Index of functions and methods 49
**set_problem_name** (*name*)

Sets the name of the SemidefiniteProgram.

**INPUT:**

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

**EXAMPLES:**

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

**show()**

Display the SemidefiniteProgram in a human-readable way.

**EXAMPLES:**

When constraints and variables have names

```python
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]
```

**slack** (*i*, **sparse=False**)

Slack of the *i*-th constraint

Available after self.solve() is called, otherwise the result is undefined

**INPUT:**

- *index* (integer) – the constraint’s id.

**OUTPUT:**

The matrix of the slack of the *i*-th constraint

**EXAMPLES:**

```python
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: # needs cvxopt
sage: p.solve()  # tol 1e-08
-3.0
sage: B1 = p.slack(1); B1  # tol 1e-08
[0.0 0.0]
[0.0 0.0]
sage: B1.is_positive_definite()
True
sage: x = sorted(p.get_values(x).values())
sage: x[0]*b1 + x[1]*b2 - b3 + B1  # tol 1e-09
[0.0 0.0]
[0.0 0.0]

solve(objective_only=False)

Solve 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.

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.

EXAMPLES:

sage: # needs cvxopt
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: N(p.solve(), 4)

Optimal solution found.

-11.

**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:**

```python
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
sage: s = p.sum(v[i] for i in range(90))
```

The following command:

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

is much more efficient than:

```python
sage: s = sum(v[i] for i in range(90))
```
CHAPTER
FOUR

LINEAR FUNCTIONS AND CONSTRAINTS

This module implements linear functions (see \texttt{LinearFunction}) in formal variables and chained (in)equalities between them (see \texttt{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)
<class 'sage.numerical.linear_functions.LinearFunction'>

sage: c = (0 <= f); c  # a constraint
  0 <= 1 + x_0 + 2*x_1
sage: type(c)
<class '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 \texttt{mip}).

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

```
sage: p = MixedIntegerLinearProgram()
sage: x = p.new_variable()
  x_0 == x_1 == x_2 == x_3
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
```
Warning: The implementation of chained inequalities uses a Python hack to make it work, so it is not completely robust. In particular, while constants are allowed, no two constants can appear next to each other. The following does not work for example:

```
sage: x[0] <= 3 <= 4
True
```

If you really need this for some reason, you can explicitly convert the constants to a `LinearFunction`:

```
sage: from sage.numerical.linear_functions import LinearFunctionsParent
sage: LF = LinearFunctionsParent(QQ)
sage: x[1] <= LF(3) <= LF(4)
x_1 <= 3 <= 4
```

---

class `sage.numerical.linear_functions.LinearConstraint`

Bases: `LinearFunctionOrConstraint`

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 == C`.

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.

**EXAMPLES:**

```
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable()
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()
```

(continues on next page)
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
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()

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, LinearConstraintsParent)
 sage: LF = LinearFunctionsParent(QQ)
 sage: LinearConstraintsParent(LF)
 Linear constraints over Rational Field
```
class sage.numerical.linear_functions.LinearConstraintsParent_class

Bases: 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: LinearFunctionOrConstraint

An elementary algebra to represent symbolic linear functions.

Warning: You should never instantiate LinearFunction manually. Use the element constructor in the parent instead.

EXAMPLES:

For example, do this:

sage: p = MixedIntegerLinearProgram()
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 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.

EXAMPLES:

```
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
```

`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.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram()
sage: LF = p.linear_functions_parent()
sage: lf = LF({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

\textbf{iteritems()}

Iterate over the index, coefficient pairs.

\textbf{OUTPUT:}

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

\textbf{EXAMPLES:}

\begin{verbatim}
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 sorted(f.iteritems()):
....:     print('id = {}, coeff = {}'.format(id, coeff))
id = -1 coeff = 1/2
id = 0 coeff = 3/2
id = 1 coeff = 3/5
\end{verbatim}

\textbf{class}  \texttt{sage.numerical.linear_functions.LinearFunctionOrConstraint}

\textbf{Bases:} \texttt{ModuleElement}

Base class for \texttt{LinearFunction} and \texttt{LinearConstraint}.

This class exists solely to implement chaining of inequalities in constraints.

\texttt{sage.numerical.linear_functions.LinearFunctionsParent()}

Return the parent for linear functions over \texttt{base_ring}.

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

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{base_ring} – a ring. The coefficient ring for the linear functions.
\end{itemize}

\textbf{OUTPUT:}

The parent of the linear functions over \texttt{base_ring}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.numerical.linear_functions import LinearFunctionsParent
sage: LinearFunctionsParent(QQ)
Linear functions over Rational Field
\end{verbatim}

\textbf{class}  \texttt{sage.numerical.linear_functions.LinearFunctionsParent_class}

\textbf{Bases:} \texttt{Parent}

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

\textbf{Warning:} You should use \texttt{LinearFunctionsParent()} to construct instances of this class.

\textbf{INPUT/OUTPUT:}

See \texttt{LinearFunctionsParent()}
EXAMPLES:

```python
sage: from sage.numerical.linear_functions import LinearFunctionsParent_class
sage: LinearFunctionsParent_class
<class '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:**

```python
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:**

```python
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
```
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 $\mathbb{R}$-module of linear functions and the $\mathbb{R}$-vector/matrix space, with the latter viewed as an $\mathbb{R}$-module ($\mathbb{R}$ is usually $\mathbb{Q}$ or $\mathbb{R}$DF 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:

\begin{verbatim}
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 ]
\end{verbatim}

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 \texttt{dict()} method. For convenience, you can also retrieve a single coefficient with \texttt{coefficient()}. For example:

\begin{verbatim}
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])
\end{verbatim}

(continues on next page)
Just like `sage.numerical.linear_functions`, (in)equality becomes symbolic inequalities. See `linear_tensor_constraints` for details.

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

```python
sage.numerical.linear_tensor.LinearTensorParent(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:**

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

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

**Bases:** `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:**

```python
from sage.numerical.linear_tensor import LinearTensorParent_class
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:**

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

```python
mip = MixedIntegerLinearProgram()
LF = mip.linear_functions_parent()
LF.tensor(RDF^2).is_matrix_space()
False
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:**

```python
```
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: 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
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)
<class 'sage.numerical.linear_tensor_element.LinearTensor'>
```

```python
class sage.numerical.linear_tensor_element.LinearTensor
    Bases: 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
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
```

```python
coefficient(x)

    Return one of 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.
```

EXAMPLES:

```sage
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)
```

(continues on next page)
sage: lt.coefficient(4)
(0.0, 0.0)
sage: lt.coefficient(-1)
(-5.0, -5.0)

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.

EXAMPLES:

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:

```python
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:

```python
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)
```

```python
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]
```

```python
class sage.numerical.linear_tensor_constraints.LinearTensorConstraint
```

Bases: `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.
EXAMPLES:

```python
sage: mip.<b> = MixedIntegerLinearProgram()
(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:**

```python
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:**

```python
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:**

```python
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()

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....: →LinearTensorConstraintsParent, LinearTensorConstraintsParent
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_parent)

Bases: 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

(continues on next page)
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
8.1 Functions and Methods

`sage.numerical.optimize.binpacking(items, maximum, k=1, solver=None, verbose=None, integrality_tolerance=0)`

Solve 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 packed in the bins, while ensuring that the sum of the weights of the items packed in each bin is at most $k$?

For more informations, see Wikipedia article Bin_packing_problem.

Two versions of this problem are solved by this algorithm:

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

**INPUT:**

- `items` – list or dict; either a list of real values (the items’ weight), or a dictionary associating to each item its weight.
- `maximum` – (default: 1); the maximal size of a bin
- `k` – integer (default: None); 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 possible number of bins.
- `solver` – (default: None) Specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to None, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.
OUTPUT:
A list of lists, each member corresponding to a bin and containing either the list of the weights inside it when items is a list of items' weight, or the list of items inside it when items is a dictionary. If there is no solution, an exception is raised (this can only happen when \( k \) is specified or if \( \text{maximum} \) is less than the weight 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:

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

Checking the bins are of correct size

```python
sage: all(sum(b) <= 1 for b in bins)  # needs sage.numerical.mip
True
```

Checking every item is in a bin

```python
sage: b1, b2, b3 = bins  # needs sage.numerical.mip
sage: all((v in b1 or v in b2 or v in b3) for v in values)  # needs sage.numerical.mip
True
```

And only in one bin

```python
sage: sum(len(b) for b in bins) == len(values)  # needs sage.numerical.mip
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

```python
sage: from sage.numerical.optimize import binpacking
sage: binpacking([0.2,0.3,0.8,0.9], k=2)  # needs sage.numerical.mip
Traceback (most recent call last):
  ... ValueException: this problem has no solution
```

We can also provide a dictionary keyed by items and associating to each item its weight. Then, the bins contain the name of the items inside it

```python
sage: values = {'a':1/5, 'b':1/3, 'c':2/3, 'd':3/4, 'e':5/7}
sage: bins = binpacking(values)  # needs sage.numerical.mip
sage: set(flatten(bins)) == set(values.keys())  # needs sage.numerical.mip
True
```
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 \([x_1, x_2, \ldots, x_k, f_1], [x_2, x_2, \ldots, x_k, f_2], \ldots, [x_n, x_n, \ldots, x_k, f_n]\) 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:

```python
sage: set_random_seed(0)
sage: data = [(i, 1.2 * sin(0.5*i-0.2) + 0.1 * normalvariate(0, 1)) for i in xsrange(0, 4*pi, 0.2)]
```

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

```python
sage: var('a, b, c, x')

(a, b, c, x)
```

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

```python
sage: find_fit(data, model) #...
```

We can also use a Python function for the model:

```python
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)
```

We search for a formula for the \(n\)-th prime number:
ALGORITHM:

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

```
sage: # needs sage.libs.pari
sage: dataprime = [(i, nth_prime(i)) for i in range(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..., 0.85762501456...)
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 $f$.
- tol – the convergence tolerance
- maxfun – maximum function evaluations

**OUTPUT:**

- minval – (float) the minimum value that $f$ takes on in the interval $[a, b]$
- x – (float) the point at which $f$ 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..., 3.42557079030572...)
```
Continuing from previous page:

```plaintext
sage: find_local_minimum(f, 1, 5, tol=1e-2, maxfun=10)
(-3.28837084598..., 3.4250840220...)
sage: show(plot(f, 0, 20))
˓→ needs sage.plot
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 [github issue #2607](https://github.com/sagemath/sage/issues/2607).

```plaintext
sage: # needs sage.symbolic
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:

```plaintext
sage: # needs sage.symbolic
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):

```plaintext
sage: # needs sage.plot sage.symbolic
sage: plot(f, (x, -2.5, -1)).ymin()
-2.182...
sage: plot(f, (x, -2.5, 2)).ymin()
-2.182...
```

**ALGORITHM:**

Uses `scipy.optimize.fminbound` which uses Brent’s method.

**AUTHOR:**

- William Stein (2007-12-07)

```plaintext
sage.numerical.optimize.find_root (f, a, b, xtol=1e-12, rtol=8.881784197001252e-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 $\geq 0$. The routine modifies this to take into account the relative precision of doubles. By default, rtol is $4*numpy.finfo(float).eps$, the minimum allowed value for `scipy.optimize.brentq` which is what this method uses underneath. This value is equal to $2.0**-50$ for IEEE-754 double precision floats as used by Python.
- $maxiter$ – integer; if convergence is not achieved in $maxiter$ iterations, an error is raised. Must be $\geq 0$.
- $full_output$ – bool (default: False), if True, also return object that contains information about convergence.

---

**8.1. Functions and Methods**
EXAMPLES:

An example involving an algebraic polynomial function:

```python
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)  # abs tol 1e-6 (note -- precision of answer isn't very...
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 \( x \geq 2.01 \):

```python
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:

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

This agrees with the plot:

```python
sage: plot(f,2,2.01)
```

The following example was added due to github issue #4942 and demonstrates that the function need not be defined at the endpoints:

```python
sage: find_root(x^2*log(x,2)-1,0, 2)  # abs tol 1e-6
1.41421356237
```

The following is an example, again from github issue #4942 where Brent’s method fails. Currently no other method is implemented, but at least we acknowledge the fact that the algorithm fails:

```python
sage: find_root(1/(x-1)+1,0, 2)
0.0
sage: find_root(1/(x-1)+1,0.00001, 2)
Traceback (most recent call last):
  ... implem error: Brent's method failed to find a zero for f on the interval
```

An example of a function which evaluates to NaN on the entire interval:

```python
sage: f(x) = 0.0 / max(0, x)
sage: find_root(f, -1, 0)
Traceback (most recent call last):
  ... ime l error: f appears to have no zero on the interval
```

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' – using the downhill simplex algorithm
  - 'powell' – use the modified Powell algorithm
  - 'bfgs' – (Broyden-Fletcher-Goldfarb-Shanno) requires gradient
  - 'cg' – (conjugate-gradient) requires gradient
  - 'ncg' – (newton-conjugate gradient) requires gradient and hessian
- **verbose** – (optional, default: False) print convergence message

**Note:** For additional information on the algorithms implemented in this function, consult SciPy’s documentation on optimization and root finding

**EXAMPLES:**

Minimize a fourth order polynomial in three variables (see the Wikipedia article Rosenbrock_function):

```sage
vars = var('x y z')
# needs sage.symbolic
f = 100*(y-x^2)^2 + (1-x)^2 + 100*(z-y^2)^2 + (1-y)^2
# needs sage.symbolic
minimize(f, [.1,.3,.4]) # abs tol 1e-6
# needs sage.symbolic
(1.0, 1.0, 1.0)
```

Try the newton-conjugate gradient method; the gradient and hessian are computed automatically:

```sage
minimize(f, [.1,.3,.4], algorithm="ncg") # abs tol 1e-6
# needs sage.symbolic
(1.0, 1.0, 1.0)
```

We get additional convergence information with the `verbose` option:

```sage
minimize(f, [.1,.3,.4], algorithm="ncg", verbose=True)
# needs sage.symbolic
Optimization terminated successfully.
...
(0.999999..., 0.999999..., 0.999999...)
```

Same example with just Python functions:
```
sage: def rosen(x):  # The Rosenbrock function
....:     return sum(100.0*(x[1r:-1r]**2.0) + (1r-x[1r:-1r])**2.0)
sage: minimize(rosen, [.1,.3,.4])  # abs tol 3e-5
(1.0, 1.0, 1.0)
```

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

```
sage: def rosen(x):  # The Rosenbrock function
....:     return sum(100.0*(x[1r:-1r]**2.0) + (1r-x[1r:-1r])**2.0)
sage: import numpy
sage: from numpy import zeros
sage: def rosen_der(x):
....:     xm = x[1r:-1r]
....:     xm_m1 = x[1r:-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[0r])
....:     der[-1] = 200r*(x[-1r]-x[-2r]**2r)
....:     return der
sage: minimize(rosen, [.1,.3,.4], gradient=rosen_der,
....:     # abs tol 1e-6
....:     algorithm="bfgs")
(1.0, 1.0, 1.0)
```

```
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 \((\text{None, None})\).
- `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 [ZBN1997].
- `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: 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 $\sin (\pi x y)$:

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

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

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

Rosenbrock function (see the Wikipedia article Rosenbrock_function):

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

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 'view(...)' or '%display typeset' for details)
```

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

$$
\begin{align*}
\text{max} \quad & 10C + 5B \\
\text{subject to} \quad & C + B \leq 1000 \\
\text{and} \quad & 3C + B \leq 1500 \\
\text{and} \quad & C, B \geq 0
\end{align*}
$$
Since it has only two variables, we can solve it graphically:

```
sage: P.plot()
needs sage.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 ...)
```
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*}
\ldots
\end{equation*}
\text{The optimal value: } \$6250\$. \text{ 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 ...)
```

Using typeset mode as recommended, you’ll see

```
x_3 = 1000 - C - B
x_4 = 1500 - 3C - B
z = 0 + 10C + 5B
```

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:
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

```python
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, objective_constant_term=0)
```

**Bases:** 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
- objective_constant_term – (default: 0) a constant term of the objective

EXAMPLES:

We will construct the following problem:

\[
\begin{align*}
\text{max} & \quad 10C + 5B \\
& \quad C + B \leq 1000 \\
& \quad 3C + B \leq 1500 \\
& \quad C, B \geq 0 
\end{align*}
\]

```
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()
\[
\begin{bmatrix}
1 & 1 \\
3 & 1
\end{bmatrix}
\]
sage: P.A()
\[
\begin{bmatrix}
1 & 1 \\
3 & 1
\end{bmatrix}
\]

**Abcx()**

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

**OUTPUT:**

- a tuple

**EXAMPLES:**

```sage
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))
```

**add_constraint**(coefficients, constant_term, constraint_type='<=')

Return a new LP problem by adding a constraint to `self`.

**INPUT:**

- coefficients – coefficients of the new constraint
- constant_term – a constant term of the new constraint
- constraint_type – (default: "<=") a string indicating the constraint type of the new constraint

**OUTPUT:**

- an LP problem

**EXAMPLES:**

```sage
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (10, 5)
sage: P = InteractiveLPProblem(A, b, c)
sage: P1 = P.add_constraint(([2, 4]), 2000, "<=")
sage: P1.Abcx()
([1 1]
 [3 1], (1000, 1500, 2000), (10, 5), (x1, x2))
sage: P1.constraint_types()
('('<=', '<=', '<=')
```
sage: P.Abcx()
(1 1
[3 1], (1000, 1500), (10, 5), (x1, x2)
)
sage: P.constraint_types()
('<=', '<=')
sage: P2 = P.add_constraint(([2, 4, 6]), 2000, "<=")
Traceback (most recent call last):
...  
TypeError: number of columns must be the same, not 2 and 3
sage: P3 = P.add_constraint(([2, 4]), 2000, "<")
Traceback (most recent call last):
...  
ValueError: unknown constraint type

\texttt{b()}

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

\textbf{OUTPUT:}

- a vector

\textbf{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)

\texttt{base\_ring()}

Return the base ring of self.

\textbf{Note:} The base ring of LP problems is always a field.

\textbf{OUTPUT:}

- a ring

\textbf{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:

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

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

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

constraint_types()  

Return a tuple listing the constraint types of all rows.

OUTPUT:

```python
9.1. Classes and functions 89
```
• a tuple of strings

EXAMPLES:

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

decision_variables()
Return decision variables of self, i.e. $x$.

OUTPUT:
• a vector

EXAMPLES:

```python
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:

```python
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
```

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

Note that infeasible problems are always bounded:

sage: b = (-1000, 1500)
sage: P = InteractiveLPProblem(A, b, c, variable_type=">=")
sage: P.is_feasible()
False
sage: P.is_bounded()
True

is_feasible(*x)
Check if self or given solution is feasible.

INPUT:
• (optional) anything that can be interpreted as a valid solution for this problem, i.e. a sequence of values for all decision variables

OUTPUT:
• True is this problem or given solution 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, variable_type=">=")
sage: P.is_feasible()
True
sage: P.is_feasible(100, 200)
True
sage: P.is_feasible(1000, 200)
False
sage: P.is_feasible([1000, 200])
False

(continues on next page)
sage: P.is_feasible(1000)
Traceback (most recent call last):
...
TypeError: given input is not a solution for this problem

is_negative()

Return True when the problem is of type "-max" or "-min".

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_negative()
False
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=", problem_type="-min")
sage: P.is_negative()
True

is_optimal(*x)

Check if given solution is feasible.

INPUT:

• anything that can be interpreted as a valid solution for this problem, i.e. a sequence of values for all decision variables

OUTPUT:

• True is the given solution is optimal, False otherwise

EXAMPLES:

sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
sage: c = (15, 5)
sage: P = InteractiveLPProblem(A, b, c, variable_type=">=")
sage: P.is_optimal(100, 200)
False
sage: P.is_optimal(500, 0)
True
sage: P.is_optimal(499, 3)
True
sage: P.is_optimal(501, -3)
False

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="\ge")
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="\ge")
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="\ge")
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="\ge")
sage: P.n_constraints()
(continues on next page)
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)```

objective_constant_term()
Return the constant term of the objective.

OUTPUT:
• a number

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_constant_term()
0
sage: P.optimal_value()
6250
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"],
....: variable_type=">=", objective_constant_term=-1250)```
objective_value(*x*)

Return the value of the objective on the given solution.

INPUT:

- anything that can be interpreted as a valid solution for this problem, i.e. a sequence of values for all decision variables

OUTPUT:

- the value of the objective on the given solution taking into account `objective_constant_term()` and `is_negative()`

EXAMPLES:

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblem(A, b, c, variable_type=">=")
P.objective_value(100, 200)
```

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])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=")
P.optimal_solution()
```

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

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

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

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=">=")  
```
In this case the plot works better with the following axes ranges:

```python
sage: p = P.plot_feasible_set(0, 1000, 0, 1500)  # needs sage.plot
sage: p.show()  # needs sage.plot
```

**problem_type()**

Return the problem type.

Needs to be used together with `is_negative`.

**OUTPUT:**

- a string, one of "max", "min".

**EXAMPLES:**

```python
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.problem_type()
'max'
```

```python
sage: P = InteractiveLPProblem(A, b, c, ["C", "B"], variable_type=">=",
    problem_type="-min")

sage: P.problem_type()
'min'
```

**standard_form** *(transformation=False, **kwds)*

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

**INPUT:**

- `transformation` – (default: False) if True, a map converting solutions of the problem in standard form to the original one will be returned as well

- you can pass (as keywords only) `slack_variables`, `auxiliary_variable`, "objective_name" to the constructor of `InteractiveLPProblemStandardForm`

**OUTPUT:**

- an `InteractiveLPProblemStandardForm` by itself or a tuple with variable transformation as the second component

**EXAMPLES:**

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

(continues on next page)
variable_types()

Return a tuple listing the variable types of all decision variables.

OUTPUT:

• a tuple of strings

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.variable_types()
('>=', '')
Return decision variables of `self`, i.e. \( x \).

**OUTPUT:**

- a vector

**EXAMPLES:**

```python
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)
```

```python
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_pimal=True, objective_name=None, objective_constant_term=0)
Bases: 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:

\[
\pm \max c^T x \\
A^T x \leq b \\
x \geq 0
\]

**INPUT:**

- \( A \) – a matrix of constraint coefficients
- \( b \) – a vector of constraint constant terms

9.1. Classes and functions 99
### InteractiveLPProblemStandardForm

- **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 string 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()`
- **objective_constant_term** – (default: 0) a constant term of the objective

#### EXAMPLES:

```python
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:

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

#### add_constraint `(coefficients, constant_term, slack_variable=None)`

Return a new LP problem by adding a constraint to `self`.

**INPUT:**
- **coefficients** – coefficients of the new constraint
- **constant_term** – a constant term of the new constraint
- **slack_variable** – (default: depends on `style()`) a string giving the name of the slack variable of the new constraint

**OUTPUT:**
- an LP problem in standard form

#### EXAMPLES:

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

(continues on next page)
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

\[
\begin{align*}
\max & -x_0 \\
-x_0 + A_i x & \leq b_i \text{ for all } i \\
\end{align*}
\]

\[x \geq 0\]

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:

```python
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:

```python
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:
**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:**

```python
sage: A = ([1, 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

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

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
```

**static random_element** *(m, n, bound=5, special_probability=0.2, **kwds)*

Construct a random `InteractiveLPProblemStandardForm`.

**INPUT:**

* m – the number of constraints/basic variables
* n – the number of decision/non-basic variables
* bound – (default: 5) a bound on coefficients
* special_probability – (default: 0.2) probability of constructing a problem whose initial dictionary is allowed to be primal infeasible or dual feasible

All other keyword arguments are passed to the constructor.

**EXAMPLES:**

```
sage: InteractiveLPProblemStandardForm.random_element(3, 4)
LP problem (use view(...) or %display typeset for details)
```

**revised_dictionary** *(\*\_B)*

Construct a revised dictionary for self.

**INPUT:**

9.1. Classes and functions 105
• 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*}
(continues on next page)
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_object
Abstract base class for dictionaries for LP problems.
Instantiating this class directly is meaningless, see LPDictionary and LPRevisedDictionary for useful extensions.

add_row(nonbasic_coefficients, constant, basic_variable=None)
Return a dictionary with an additional row based on a given dictionary.

INPUT:
• nonbasic_coefficients—a list of the coefficients for the new row (with which nonbasic variables are subtracted in the relation for the new basic variable)
• constant—the constant term for the new row
• basic_variable—(default: depends on style()) a string giving the name of the basic variable of the new row

OUTPUT:
• a new dictionary of the same class

EXAMPLES:
```
sage: A = ([1, 1, 7], [8, 2, 13], [34, 17, 12])
sage: b = (2, 17, 6)
sage: c = (55/10, 21/10, 14/30)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary("x1", "x2", "x4")
sage: D1 = D.add_row([7, 11, 19], 42, basic_variable='c')
sage: D1.row_coefficients("c")
(7, 11, 19)
```
**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 = InteractiveLPPProblemStandardForm(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 = InteractiveLPPProblemStandardForm(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)
```

**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 = InteractiveLPPProblemStandardForm(A, b, c)
sage: D = P.initial_dictionary()
sage: D.basic_variables()
```

9.1. Classes and functions
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)

\textbf{column\_coefficients\( (v) \)}

Return the coefficients of a nonbasic variable.

\textbf{INPUT:}

\begin{itemize}
  \item \(v\) – a nonbasic variable of \texttt{self}, can be given as a string, an actual variable, or an integer interpreted as the index of a variable
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item a vector of coefficients of a nonbasic variable
\end{itemize}

\textbf{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.column_coefficients(1)
(1, 3)
```

\textbf{constant\_terms\( () \)}

Return the constant terms of relations of \texttt{self}.

\textbf{OUTPUT:}

\begin{itemize}
  \item a vector.
\end{itemize}

\textbf{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)
```

\textbf{coordinate\_ring\( () \)}

Return the coordinate ring of \texttt{self}.

\textbf{OUTPUT:}

\begin{itemize}
  \item a polynomial ring in \texttt{auxiliary\_variable()}, \texttt{decision\_variables()}, and \texttt{slack\_variables()} of \texttt{self} over the \texttt{base\_ring()}
\end{itemize}

\textbf{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
```

`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)
```

`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
```

`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)
```

The same works for revised dictionaries as well:
```
sage: D = P.revised_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_name()**

Return the objective name of self.

OUTPUT:

- a symbolic expression

**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_name()
z

```
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
```

```
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 \texttt{basic_variables()}, but only \(x_i\) with positive \(a_{ik}\) are considered.

**EXAMPLES:**

```python
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)]
```

**row_coefficients**\((v)\)

Return the coefficients of the basic variable \(v\).

These are the coefficients with which nonbasic variables are subtracted in the relation for \(v\).

**INPUT:**

• \(v\) – a basic variable of \texttt{self}, can be given as a string, an actual variable, or an integer interpreted as the index of a variable

**OUTPUT:**

• a vector of coefficients of a basic variable

**EXAMPLES:**

```python
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.row_coefficients("x1")
(1/10, -1/5)
```

We can also use indices of variables:

```python
sage: D.row_coefficients(1)
(1/10, -1/5)
```

Or use variable names without quotes after injecting them:

```python
sage: P.inject_variables()
Defining x0, x1, x2, x3, x4
sage: D.row_coefficients(x1)
(1/10, -1/5)
```
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:

```python
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:

```python
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*}
\text{...}
\end{equation*}
Leaving: $x_{3}$. Entering: $x_{1}$.
\begin{equation*}
\text{...}
\end{equation*}
sage: D.is_optimal()
True
```

This method detects infeasible problems:

```python
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*}
\text{...}
\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:
EXAMPLES:

```python
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:

```python
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:

```python
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.
```

**update**

Update self using previously set entering and leaving variables.

EXAMPLES:

```python
sage: A = ([1, 1], [3, 1])
sage: b = (1000, 1500)
```
(continues on next page)
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

```python
class sage.numerical.interactive_simplex_method.LPDictionary(A, b, c, objective_value, basic_variables, nonbasic_variables, objective_name):
    Bases: LPAbstractDictionary
    Construct a dictionary for an LP problem.
    A dictionary consists of the following data:
    \[
    x_B = b - Ax_N
    \]
    \[
    z = z^* + cx_N
    \]
    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:
    ```python
    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 ...)
    ```
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
```

**add_row** *(nonbasic_coefficients, constant, basic_variable=None)*

Return a dictionary with an additional row based on a given dictionary.

**INPUT:**
- **nonbasic_coefficients**– a list of the coefficients for the new row (with which nonbasic variables are subtracted in the relation for the new basic variable)
- **constant** – the constant term for the new row
- **basic_variable** – (default: depends on style()) a string giving the name of the basic variable of the new row

**OUTPUT:**
- a dictionary

**EXAMPLES:**

```
sage: A = ([[-1, 1, 7], [8, 2, 13], [34, 17, 12]])
sage: b = (2, 17, 6)
sage: c = (55/10, 21/10, 14/30)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.dictionary("x1", "x2", "x4")
sage: D1 = D.add_row([7, 11, 19], 42, basic_variable='c')
sage: D1.row_coefficients("c")
(7, 11, 19)
sage: D1.constant_terms()[-1]
42
sage: D1.basic_variables()[-1]
c
```

**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)]
```
\textbf{column\_coefficients}(v)

Return coefficients of a nonbasic variable.

INPUT:

\begin{itemize}
  \item $v$ \textendash{} a nonbasic variable of \texttt{self}, can be given as a string, an actual variable, or an integer interpreted as the index of a variable
\end{itemize}

OUTPUT:

\begin{itemize}
  \item a vector
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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.column_coefficients(1)
(1, 3)
\end{verbatim}

\textbf{constant\_terms}()

Return the constant terms of relations of \texttt{self}.

OUTPUT:

\begin{itemize}
  \item a vector.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{nonbasic\_variables}()

Return non-basic variables of \texttt{self}.

OUTPUT:

\begin{itemize}
  \item a vector
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{objective\_coefficients}()

Return coefficients of the objective of \texttt{self}.

OUTPUT:

\begin{itemize}
  \item a vector
\end{itemize}
EXAMPLES:

```python
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_name()**

Return the objective name of `self`.

**OUTPUT:**

- a symbolic expression

**EXAMPLES:**

```python
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_name()
z
```

**objective_value()**

Return the value of the objective at the `basic_solution()` of `self`.

**OUTPUT:**

- a number

**EXAMPLES:**

```python
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
```

**static random_element**(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) # indirect doctest
LP problem dictionary (use 'view(...)' or '%display typeset' for details)

row_coefficients(v)

Return the coefficients of the basic variable v.

These are the coefficients with which nonbasic variables are subtracted in the relation for v.

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

OUTPUT:

- a vector of coefficients of a basic variable

EXAMPLES:

sage: A = ([-1, 1], [8, 2])
sage: b = (2, 17)
sage: c = (55/10, 21/10)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.final_dictionary()
sage: D.row_coefficients("x1")
(1/10, -1/5)

We can also use indices of variables:

sage: D.row_coefficients(1)
(1/10, -1/5)

Or use variable names without quotes after injecting them:

sage: P.inject_variables()
Defining x0, x1, x2, x3, x4
sage: D.row_coefficients(x1)
(1/10, -1/5)

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

9.1. Classes and functions 125
class sage.numerical.interactive_simplex_method.LPRevisedDictionary (problem, basic_variables)

Bases: 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

\[ \pm \text{max } cx \]
\[ Ax \leq b \]
\[ x \geq 0 \]

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

\[ \pm \text{max } \bar{c}\bar{x} \]
\[ \bar{A}\bar{x} = \bar{b} \]
\[ \bar{x} \geq 0 \]

and any dictionary is a system of equations equivalent to \( \bar{A}\bar{x} = \bar{b} \), 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

\[
x_B = \bar{B}^{-1} \bar{b} - \bar{B}^{-1} \bar{A}_N x_N
\]
\[
z = \bar{y} \bar{b} + (c_N - \bar{y}^T \bar{A}_N) x_N
\]

where \( y = c_B^T \bar{B}^{-1} \). 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 \( \bar{B}^{-1} \), 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 ...)
```

The same dictionary can be constructed through the problem:
When this dictionary is typeset, you will see two tables like these ones:

<table>
<thead>
<tr>
<th>$x_B$</th>
<th>$c_B$</th>
<th>$B^{-1}$</th>
<th>$y$</th>
<th>$B^{-1}b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>10</td>
<td>$-\frac{1}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{5}{2}$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>5</td>
<td>$\frac{3}{2}$</td>
<td>$-\frac{1}{2}$</td>
<td>$\frac{5}{2}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_N$</th>
<th>$x_3$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_N^T$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$y^T A_N$</td>
<td>$\frac{5}{2}$</td>
<td>$\frac{5}{2}$</td>
</tr>
<tr>
<td>$c_N^T - y^T A_N$</td>
<td>$-\frac{5}{2}$</td>
<td>$-\frac{5}{2}$</td>
</tr>
</tbody>
</table>

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:**

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

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

(continues on next page)
B()  
Return the $B$ matrix, i.e. constraint coefficients of basic variables.

OUTPUT:
• a matrix

EXAMPLES:

```python
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:

```python
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 $B_{new} = B_{old}E$.

EXAMPLES:

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

(continues on next page)
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:**

```python
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()
```

```python
[ 1 -1/3]
[ 0 1/3]
```

add\_row\_nonbasic\_coefficients, constant, basic\_variable=None)

Return a dictionary with an additional row based on a given dictionary.

The implementation of this method for revised dictionaries adds a new inequality constraint to the problem, in which the given basic variable becomes the slack variable. The resulting dictionary (with basic variable added to the basis) will have the given nonbasic coefficients and constant as a new row.

**INPUT:**

- nonbasic\_coefficients—a list of the coefficients for the new row (with which nonbasic variables are subtracted in the relation for the new basic variable)
- constant—the constant term for the new row
- basic\_variable=(default: depends on style()) a string giving the name of the basic variable of the new row

**OUTPUT:**

- a revised dictionary

**EXAMPLES:**

```python
sage: A = ([-1, 1111, 3, 17], [8, 222, 7, 6],
....: [3, 7, 17, 5], [9, 5, 7, 3])
sage: b = (2, 17, 11, 27)
sage: c = (5/133, 1/10, 1/18, 47/3)
sage: P = InteractiveLPProblemStandardForm(A, b, c)
sage: D = P.final_revised_dictionary()
```
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:**

```python
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:**

```python
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:**

```python
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)
```

**column_coefficients(v)**

Return the coefficients of a nonbasic variable.

**INPUT:**

- `v` – a nonbasic variable of self, can be given as a string, an actual variable, or an integer interpreted as the index of a variable

**OUTPUT:**

- a vector
EXAMPLES:

```python
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.column_coefficients(1)
(1, 3)
```

**constant_terms()**

Return constant terms in the relations of self.

**OUTPUT:**

- a vector

**EXAMPLES:**

```python
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:**

```python
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 ...)
```

**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:**
nonbasic_variables()

Return non-basic variables of self.

OUTPUT:
  • a vector

EXAMPLES:

```sage
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
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_name()

Return the objective name of self.

OUTPUT:
  • a symbolic expression

EXAMPLES:

```sage
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_name()
z
```
**objective_value()**

Return the value of the objective at the basic solution of self.

OUTPUT:

- a number

EXAMPLES:

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblemStandardForm(A, b, c)
D = P.revised_dictionary()
D.objective_value()
0
```

**problem()**

Return the original problem.

OUTPUT:

- an LP problem in standard form

EXAMPLES:

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblemStandardForm(A, b, c)
D = P.revised_dictionary()
D.problem() is P
True
```

**row_coefficients(v)**

Return the coefficients of the basic variable v.

These are the coefficients with which nonbasic variables are subtracted in the relation for v.

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

OUTPUT:

- a vector of coefficients of a basic variable

EXAMPLES:

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblemStandardForm(A, b, c)
D = P.revised_dictionary()
D.row_coefficients("x3")
(-1, 1)
```

We can also use indices of variables:

```sage
D.row_coefficients(3)
(-1, 1)
```
Or variable names without quotes after injecting them:

```sage
P.inject_variables()
Defining x0, x1, x2, x3, x4
D.row_coefficients(x3)
(-1, 1)
```

**update()**

Update `self` using previously set entering and leaving variables.

**EXAMPLES:**

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblemStandardForm(A, b, c)
D = P.revised_dictionary()
D.objective_value()
0
D.enter("x1")
D.leave("x4")
D.update()
D.objective_value()
5000
```

**x_B()**

Return the basic variables of `self`.

**OUTPUT:**

- a vector

**EXAMPLES:**

```sage
A = ([1, 1], [3, 1])
b = (1000, 1500)
c = (10, 5)
P = InteractiveLPProblemStandardForm(A, b, c)
D = P.revised_dictionary()
D.basic_variables()
(x3, x4)
```

**x_N()**

Return non-basic variables of `self`.

**OUTPUT:**

- a vector

**EXAMPLES:**

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

Return the \( y \) vector, the product of \( c_B() \) and \( B_{\text{inverse}}() \).

OUTPUT:

- a vector

EXAMPLES:

```sage
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)
```

\[\text{sage.numerical.interactive_simplex_method.default_variable_name}(\text{variable})\]

Return default variable name for the current \textit{style}().

INPUT:

- \text{variable} - a string describing requested name

OUTPUT:

- a string with the requested name for current style

EXAMPLES:

```sage
sage: sage.numerical.interactive_simplex_method.default_variable_name("primal \rightarrow slack")
'x'
sage: sage.numerical.interactive_simplex_method.style('Vanderbei')
'Vanderbei'
sage: sage.numerical.interactive_simplex_method.default_variable_name("primal \rightarrow slack")
'w'
sage: sage.numerical.interactive_simplex_method.style('UAlberta')
'UAlberta'
```

\[\text{sage.numerical.interactive_simplex_method.random_dictionary}(m, n, \text{bound}=5, \text{special_probability}=0.2)\]

Construct a random dictionary.

INPUT:

- \text{m} - the number of constraints/basic variables
- \text{n} - the number of decision/non-basic variables
- \text{bound} - (default: 5) a bound on dictionary entries
- \text{special_probability} - (default: 0.2) probability of constructing a potentially infeasible or potentially optimal dictionary

OUTPUT:

- an \textit{LP problem dictionary}

EXAMPLES:
sage: from sage.numerical.interactive_simplex_method]
....: import random_dictionary
sage: random_dictionary(3, 4) # indirect doctest
LP problem dictionary (use 'view(...)' or '%display typeset' for 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
    - x₁, x₂, ..., xₙ for primal decision variables
    - xₙ₊₁, xₙ₊₂, ..., xₙ₊ₘ for primal slack variables
    - y₁, y₂, ..., yₘ for dual decision variables
    - yₘ₊₁, yₘ₊₂, ..., yₘ₊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
    - x₁, x₂, ..., xₙ for primal decision variables
    - w₁, w₂, ..., wₘ for primal slack variables
    - y₁, y₂, ..., yₘ for dual decision variables
    - z₁, z₂, ..., zₙ 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(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
GAUSS-LEGENDRE INTEGRATION FOR VECTOR-VALUED FUNCTIONS

Routine to perform Gauss-Legendre integration for vector-functions.

EXAMPLES:

We verify that $\int_0^1 nx^{n-1} \, dx = 1$ for $n = 1, \ldots, 4$:

```python
sage: from sage.numerical.gauss_legendre import integrate_vector
sage: prec = 100
sage: K = RealField(prec)
sage: N = 4
sage: V = VectorSpace(K, N)
sage: f = lambda x: V([((n+1)*x^n) for n in range(N)])
sage: I = integrate_vector(f, prec)
sage: max([c.abs() for c in I-V(N*[1])])
0.00000000000000000000000000000
```

AUTHORS:

- Nils Bruin (2017-06-06): initial version
- Linden Disney-Hogg (2021-06-17): documentation and integrate_vector method changes

Note: The code here is directly based on mpmath (see http://mpmath.org), but has a highly optimized routine to compute the nodes.

```python
sage.numerical.gauss_legendre.estimate_error(results, prec, epsilon)
```

Routine to estimate the error in a list of quadrature approximations.

The method used is based on Borwein, Bailey, and Girgensohn. As mentioned in mpmath: Although not very conservative, this method seems to be very robust in practice.

The routine takes a list of vector results and, under assumption that these vectors approximate a given vector approximately quadratically, gives an estimate of the maximum norm of the error in the last approximation.

INPUT:

- results – list. List of approximations to estimate the error from. Should be at least length 2.
- prec – integer. Binary precision at which computations are happening.
- epsilon – multiprecision float. Default error estimate in case of insufficient data.

OUTPUT:

An estimate of the error.
EXAMPLES:

```
sage: from sage.numerical.gauss_legendre import estimate_error
sage: prec = 200
sage: K = RealField(prec)
sage: V = VectorSpace(K, 2)
sage: a = V([1, -1])
sage: b = V([1, 1/2])
sage: L = [a + 2^(-2^i)*b for i in [0..5]]
sage: estimate_error(L, prec, K(2^(-prec)))
2.328235...e-10
```

```
sage.numerical.gauss_legendre.integrate_vector(f, prec, epsilon=None)
```

Integrate a one-argument vector-valued function numerically using Gauss-Legendre.

This function uses the Gauss-Legendre quadrature scheme to approximate the integral $\int_0^1 f(t) \, dt$.

**INPUT:**

- $\text{prec}$ – integer. Binary precision to be used.
- $\epsilon$ – multiprecision float (default: $2^{-\text{prec}+3}$). Target error bound.

**OUTPUT:**

Vector approximating value of the integral.

**EXAMPLES:**

```
sage: from sage.numerical.gauss_legendre import integrate_vector
sage: prec = 200
sage: K = RealField(prec)
sage: V = VectorSpace(K, 2)
sage: epsilon = K(2^(-prec + 4))
sage: f = lambda t: V((1 + t^2, 1/(1 + t^2)))
sage: J = V((4/3, pi/4))
# needs sage.symbolic
sage: max(c.abs() for c in (I - J)) < epsilon
# needs sage.symbolic
True
```

We can also use complex-valued integrands:

```
sage: prec = 200
sage: Kreal = RealField(prec)
sage: K = ComplexField(prec)
sage: V = VectorSpace(K, 2)
sage: epsilon = Kreal(2^(-prec + 4))
sage: f = lambda t: V((t, K(exp(2*pi*t*K.0))))
# needs sage.symbolic
sage: I = integrate_vector(f, prec, epsilon=epsilon)
# needs sage.symbolic
sage: J = V((1/2, 0))
# needs sage.symbolic
sage: max(c.abs() for c in (I - J)) < epsilon
# needs sage.symbolic
True
```
sage.numerical.gauss_legendre.integrate_vector_N(f, prec, N=3)  
Integrate a one-argument vector-valued function numerically using Gauss-Legendre, setting the number of nodes.

This function uses the Gauss-Legendre quadrature scheme to approximate the integral \( \int_0^1 f(t) \, dt \). It is different from integrate_vector by using a specific number of nodes rather than targeting a specified error bound on the result.

**INPUT:**

- \( f \) – callable. Vector-valued integrand.
- \( \text{prec} \) – integer. Binary precision to be used.
- \( N \) – integer (default: 3). Number of nodes to use.

**OUTPUT:**

Vector approximating value of the integral.

**EXAMPLES:**

```
sage: from sage.numerical.gauss_legendre import integrate_vector_N
sage: prec = 100
sage: K = RealField(prec)
sage: V = VectorSpace(K, 1)
sage: f = lambda t: V([t])
sage: integrate_vector_N(f, prec, 4)
(0.50000000000000000000000000000)
```

**Note:** The nodes and weights are calculated in the real field with \( \text{prec} \) bits of precision. If the vector space in which \( f \) takes values is over a field which is incompatible with this field (e.g. a finite field) then a `TypeError` occurs.

sage.numerical.gauss_legendre.nodes(prec)  
Compute the integration nodes and weights for the Gauss-Legendre quadrature scheme, caching the output.

Works by calling nodes_uncached.

**INPUT:**

- \( \text{degree} \) – integer. The number of nodes. Must be 3 or even.
- \( \text{prec} \) – integer (minimal value 53). Binary precision with which the nodes and weights are computed.

**OUTPUT:**

A list of (node, weight) pairs.

**EXAMPLES:**

The nodes for the Gauss-Legendre scheme are roots of Legendre polynomials. The weights can be computed by a straightforward formula (note that evaluating a derivative of a Legendre polynomial isn’t particularly numerically stable, so the results from this routine are actually more accurate than what the values the closed formula produces):

```
sage: from sage.numerical.gauss_legendre import nodes
sage: L1 = nodes(24, 53)
sage: P = RR['x'](sage.functions.orthogonal_polys.legendre_P(24, x))  # needs sage.symbolic
sage: Pdiff = P.diff()  # needs sage.symbolic
sage: L2 = [(r + 1)/2, 1/(1 - r^2)/Pdiff(r)^2]
```
(continues on next page)
\[\text{sage.numerical.gauss_legendre.nodes_uncached}(\text{degree}, \text{prec})\]

Compute the integration nodes and weights for the Gauss-Legendre quadrature scheme.

We use the recurrence relations for Legendre polynomials to compute their values. This is a version of the algorithm that in [Neu2018] is called the REC algorithm.

**INPUT:**

- \text{degree} – integer. The number of nodes. Must be 3 or even.
- \text{prec} – integer (minimal value 53). Binary precision with which the nodes and weights are computed.

**OUTPUT:**

A list of (node, weight) pairs.

**EXAMPLES:**

The nodes for the Gauss-Legendre scheme are roots of Legendre polynomials. The weights can be computed by a straightforward formula (note that evaluating a derivative of a Legendre polynomial isn't particularly numerically stable, so the results from this routine are actually more accurate than what the values the closed formula produces):

\[
\text{sage: from sage.numerical.gauss_legendre import nodes_uncached}
\]
\[
\text{sage: L1 = nodes_uncached(24, 53)}
\]
\[
\text{sage: P = RR['x'](sage.functions.orthogonal_polys.legendre_P(24, x))} \quad \#...
\]
\[
\text{sage: Pdif = P.diff()} \quad \#...
\]
\[
\text{sage: L2 = [(r + 1)/2, 1/(1 - r^2)/Pdif(r)^2)} \quad \#...
\]
\[
\text{sage: all((a[0] - b[0]).abs() < 1e-15 and (a[1] - b[1]).abs() < 1e-9} \quad \#...
\]
\[
\text{sage: all((a[0] - b[0]).abs() < 1e-15 and (a[1] - b[1]).abs() < 1e-9} \quad \#...
\]
\[
\text{sage: all((a[0] - b[0]).abs() < 1e-15 and (a[1] - b[1]).abs() < 1e-9} \quad \#...
\]
\[
\text{True}
\]

**Todo:** It may be worth testing if using the Arb algorithm for finding the nodes and weights in \text{arb/acb_calc/integrate_gl_auto_deg.c} has better performance.
LINEAR OPTIMIZATION (LP) AND MIXED INTEGER LINEAR OPTIMIZATION (MIP) SOLVER BACKENDS

11.1 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 `Solver-Name(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

```python
class sage.numerical.backends.generic_backend.GenericBackend
    Bases: SageObject

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.
```

EXAMPLES:
```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.nrows()
0
```
Numerical Optimization, Release 10.3

(continued from previous page)

```python
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(list(range(5)), list(range(5)))
sage: p.nrows()
5
```

**add_linear_constraint** \((\text{coefficients, lower_bound, upper_bound, name=None})\)

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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 2.0, 2.0)
sage: p.row(0)
([0, 1, 2, 3, 4], [0.0, 1.0, 2.0, 3.0, 4.0])
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_constraint_vector** \((\text{degree, coefficients, lower_bound, upper_bound, name=None})\)

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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
```

(continues on next page)
add_linear_constraints \( \text{(number, lower\_bound, upper\_bound, names=None)} \)

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)

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
sage: p.add_variables(5)
5
```

```python
sage: p.add_linear_constraints(5, None, 2)
```

```python
sage: p.row(4)
([], [])
```

```python
sage: p.row_bounds(4)
(None, 2.0)
```

add_variable \( \text{(lower\_bound=0, upper\_bound=None, binary=False, continuous=True, integer=False, obj=None, name=None)} \)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both positive and real.

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 integral (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

EXAMPLES:
add_variables(n, lower_bound=False, 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 nonnegative 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.

EXAMPLES:
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 \texttt{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 \texttt{solver_parameter()}).

**Note:** Has no meaning unless \texttt{solve} has been called before.

EXAMPLES:

```python  
# optional - nonexistent_lp_solver  
sage: from sage.numerical.backends.generic_backend import get_solver  
sage: p = get_solver(solver="Nonexistent_LP_solver")  
sage: p.add_variable()  
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_bounds \((\text{index})\)  

Return the bounds of a specific variable.

**INPUT:**

- \texttt{index} (integer) – the variable’s id.

**OUTPUT:**

A pair \((\text{lower_bound}, \text{upper_bound})\). Each of them can be set to \texttt{None} if the variable is not bounded in the corresponding direction, and is a real value otherwise.

**EXAMPLES:**

```python  
# optional - nonexistent_lp_solver  
sage: from sage.numerical.backends.generic_backend import get_solver  
sage: p = get_solver(solver="Nonexistent_LP_solver")  
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 column name

**INPUT:**

- index (integer) – the column id
- name (char *) – its name. When set to NULL (default), the method returns the current name.

**EXAMPLES:**

```
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: b = p.new_variable()
sage: p.set_objective(b[1] + b[2])
sage: copy(p).solve()
6.0
```

**copy()**

Returns a copy of self.

**EXAMPLES:**

```
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: b = p.new_variable()
sage: p.set_objective(b[1] + b[2])
sage: copy(p).solve()
6.0
```

**get_objective_value()**

Return the value of the objective function.

**Note:** Behavior is undefined unless solve has been called before.

**EXAMPLES:**

```
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: p.add_variable(name="I am a variable")
sage: p.col_name(0)
'I am a variable'
```

**get_relative_objective_gap()**

Return the relative objective gap of the best known solution.
For a minimization problem, this value is computed by \((\text{bestinteger} - \text{bestobjective})/(1e - 10 + |\text{bestobjective}|)\), where bestinteger is the value returned by \(\text{get_objective_value()}\) and bestobjective is the value returned by \(\text{best_known_objective_bound()}\). For a maximization problem, the value is computed by \((\text{bestobjective} - \text{bestinteger})/(1e - 10 + |\text{bestobjective}|)\).

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

### EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: b = p.new_variable(binary=True)
sage: for u,v in graphs.CycleGraph(5).edges(labels=False):
    ....:     p.add_constraint(b[u]+b[v]<=1)
sage: p.set_objective(p.sum(b[x] for x in range(5)))
sage: p.solve()
2.0
sage: pb = p.get_backend()
sage: pb.get_objective_value()
2.0
sage: pb.get_relative_objective_gap()
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.

### EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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)
0.0
sage: p.get_variable_value(1)
1.5
```

#### is_maximization ()
Test whether the problem is a maximization

### EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
```

(continues on next page)
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False

is_slack_variable_basic(index)

Test whether the slack variable of the given row is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise
an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

sage: # optional - nonexistent_LP_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
....: solver="Nonexistent_LP_solver")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method...
  → here
sage: b.solve()
0
sage: b.is_slack_variable_basic(0)
True
sage: b.is_slack_variable_basic(1)
False

is_slack_variable_nonbasic_at_lower_bound(index)

Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise
an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

sage: # optional - nonexistent_LP_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
....: solver="Nonexistent_LP_solver")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method...
  → here
sage: b.solve()
is_variable_basic(index)
Test whether the given variable is basic.
This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:
- index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
....:                     solver="Nonexistent_LP_solver")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method.
˓→here
sage: b.solve()
0
sage: b.is_variable_basic(0)
True
sage: b.is_variable_basic(1)
False
```

is_variable_binary(index)
Test whether the given variable is of binary type.

INPUT:
- index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols() 0
sage: p.add_variable() 0
sage: p.set_variable_type(0,1)
sage: p.is_variable_integer(0) True
```

is_variable_nonbasic_at_lower_bound (index)
Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols() 0
sage: p.add_variable() 0
sage: p.set_variable_type(0,1)
sage: p.is_variable_nonbasic_at_lower_bound(0) True
```
ncols()

Return the number of columns/variables.

EXAMPLES:

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLES:

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.nrows()
0
sage: p.add_linear_constraints(2, 2.0, None)
1
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

EXAMPLES:

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0.0
sage: p.objective_coefficient(0,2)
1.0
```
**objective_constant_term** (*d=\text{None}* )

Set or get the constant term in the objective function

**INPUT:**

- *d* (double) – its coefficient. If *None* (default), return the current value.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
0.0
sage: p.objective_constant_term(42)
sage: p.objective_constant_term()
42.0
```

**problem_name** (*name=\text{None}* )

Return or define the problem’s name

**INPUT:**

- *name* (str) – the problem’s name. When set to *None* (default), the method returns the problem’s name.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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.

**INPUT:**

- *i* – index of the constraint to remove.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: v = p.new_variable(nonnegative=True)
sage: x,y = v[0], v[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
sage: p.remove_constraint(0)
sage: p.solve()
10.0
sage: p.get_values([x,y])
[0.0, 3.0]
```
**remove_constraints**(constraints)

Remove several constraints.

**INPUT:**

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

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variables(2)

sage: p.add_linear_constraint([(0, 2), (1, 3)], None, 6)
sage: p.add_linear_constraint([(0, 3), (1, 2)], None, 6)

sage: p.remove_constraints([0, 1])
```

**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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variables(5)

sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)

sage: p.row(0)

4
sage: p.row_bounds(0)

(row(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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variables(5)

(continues on next page)"
row_name (index)

Return the index th row name

INPUT:

• index (integer) – the row's id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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 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)

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
[p.objective_coefficient(x) for x in range(5)]
[1.0, 1.0, 2.0, 1.0, 3.0]
```

Constants in the objective function are respected:

```python
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(solver='Nonexistent_LP_solver')
sage: x,y = p[0], p[1]
sage: p.add_constraint(2*x + 3*y, max=6)
sage: p.set_objective(x + y + 7)
sage: p.solve()
9.0
```

set_sense (sense)

Set the direction (maximization/minimization).
INPUT:

• `sense` (integer):
  
  – +1 => Maximization
  – -1 => Minimization

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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 Continuous

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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 log (verbosity) level

INPUT:

• `level` (integer) – From 0 (no verbosity) to 3.

EXAMPLES:

```python
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
```

11.1. Generic Backend for LP solvers
solve()
Solve the problem.

**Note:** This method raises MIPSolverException exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLES:

```sage
# optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(list(range(5)), list(range(5)))
sage: p.solve()
0
sage: p.objective_coefficient(0, 1)
sage: p.solve()
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().

EXAMPLES:

```sage
# optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.solver_parameter("timelimit")
sage: p.solver_parameter("timelimit", 60)
sage: p.solver_parameter("timelimit")
```

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.

EXAMPLES:

```sage
# optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

(continues on next page)
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 no upper bound. When set to False (default), the method returns the current value.

EXAMPLES:

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

write_lp (name)

Write the problem to a .lp file

INPUT:

• filename (string)

EXAMPLES:

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(2)
sage: p.add_linear_constraint([(0, 1), (1, 2)], None, 3)
sage: p.set_objective([2, 5])
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".lp") as f:
    ....:     p.write_lp(f.name)

write_mps (name, modern)

Write the problem to a .mps file

INPUT:

• filename (string)

EXAMPLES:
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variables(2)
2
sage: p.add_linear_constraint(
    [(0, 1), (1, 2)], None, 3)

sage: p.set_objective([2, 5])

sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".lp") as f:
    p.write_lp(f.name)

zero()

sage.numerical.backends.generic_backend.default_mip_solver(solver=None)

Returns/sets the default MILP solver used by Sage.

INPUT:

- solver – one of the following:
  - a string indicating one of the available solvers (see MixedIntegerLinearProgram);
  - a callable (typically a subclass of sage.numerical.backends.generic_backend.GenericBackend);
  - None (default), in which case the current default solver is returned; this is either a string or a callable.

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.

EXAMPLES:

sage: former_solver = default_mip_solver()
sage: default_mip_solver("GLPK")
'Glpk'
sage: default_mip_solver() 'Glpk'
sage: default_mip_solver("PPL")
'Ppl'
sage: default_mip_solver("GUROBI") # random
Traceback (most recent call last):
  ... ValueError: Gurobi is not available. Please refer to the documentation to install...
  ...
  ... it.
  sage: default_mip_solver("Yeahhhhhhhhhhh")
  ... ValueError: 'solver' should be set to ...
sage: default_mip_solver(former_solver)

sage.numerical.backends.generic_backend.get_solver(constraint_generation=False,
solver=None, base_ring=None)

Return a solver according to the given preferences.

INPUT:

- solver – one of the following:
  - a string indicating one of the available solvers (see MixedIntegerLinearProgram);
None (default), in which case the default solver is used (see \texttt{default_mip_solver()});
or a callable (such as a class), in which case it is called, and its result is returned.

- \texttt{base\_ring} – If not \texttt{None}, request a solver that works over this (ordered) field. If \texttt{base\_ring} is not a field, its fraction field is used.
  
  For example, is \texttt{base\_ring=ZZ} is provided, the solver will work over the rational numbers. This is unrelated to whether variables are constrained to be integers or not.

- \texttt{constraint\_generation} – Only used when \texttt{solver=None}.
  
  When set to True, after solving the \texttt{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. (Coin and SCIP are such solvers.)
  
  – Defaults to False.

See also:

- \texttt{default_mip_solver()} – Returns/Sets the default MIP solver.

\textbf{EXAMPLES:}

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver()
sage: p = get_solver(base_ring=RDF)
sage: p.base_ring()
Real Double Field
sage: p = get_solver(base_ring=QQ); p
<...sage.numerical.backends.ppl_backend.PPLBackend...>
sage: p = get_solver(base_ring=ZZ); p
<...sage.numerical.backends.ppl_backend.PPLBackend...>
sage: p.base_ring()
Rational Field
sage: p = get_solver(base_ring=AA); p
# needs sage.rings.number_field
<...sage.numerical.backends.interactivelp_backend.InteractiveLPBackend...>
sage: p.base_ring()
# needs sage.rings.number_field
Algebraic Real Field
sage: d = polytopes.dodecahedron()
sage: p = get_solver(base_ring=d.base_ring()); p
<...sage.numerical.backends.interactivelp_backend.InteractiveLPBackend...>
sage: p.base_ring()
Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.
# 236067977499790
sage: p = get_solver(solver='InteractiveLP', base_ring=QQ); p
<...sage.numerical.backends.interactivelp_backend.InteractiveLPBackend...>
sage: p.base_ring()
Rational Field
```

Passing a callable as the \texttt{solver}:

```
sage: from sage.numerical.backends.glpk_backend import GLPKBackend
sage: p = get_solver(solver=GLPKBackend); p
<...sage.numerical.backends.glpk_backend.GLPKBackend...>
```

Passing a callable that customizes a backend:

```
```
11.2 InteractiveLP Backend

AUTHORS:

- Nathann Cohen (2010-10) : generic_backend template
- Matthias Koeppe (2016-03) : this backend

class sage.numerical.backends.interactivelp_backend.InteractiveLPBackend
    Bases: GenericBackend

MIP Backend that works with InteractiveLPProblem.

This backend should be used only for linear programs over general fields, or for educational purposes. For fast computations with floating point arithmetic, use one of the numerical backends. For exact computations with rational numbers, use backend 'PPL'.

There is no support for integer variables.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
```

```
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.
```

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.ncols()  
0
sage: p.nrows()  
0
sage: p.add_linear_constraints(5, 0, None)
```

(continues on next page)
add_linear_constraint\( (\text{coefficients, lower\_bound, upper\_bound, name=None}) \)

Add a linear constraint.

INPUT:

- \text{coefficients} – an iterable of pairs \((i, v)\). In each pair, \(i\) is a variable index (integer) and \(v\) is a value (element of \text{base\_ring()}).
- \text{lower\_bound} – element of \text{base\_ring()} or None. The lower bound.
- \text{upper\_bound} – element of \text{base\_ring()} or None. The upper bound.
- \text{name} – string or None. Optional name for this row.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
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)
sage: p.add_linear_constraint( zip(range(5), range(5)), 1, 1, name='foo')
sage: p.row_name(1)
'foo'
```

add_variable\( (\text{lower\_bound=0, upper\_bound=None, binary=False, continuous=True, integer=False, obj=None, name=None, coefficients=None}) \)

Add a variable.

This amounts to adding a new column to the matrix. By default, the variable is both nonnegative and real.

In this backend, variables are always continuous (real). If integer variables are requested via the parameters \text{binary} and \text{integer}, an error will be raised.

INPUT:

- \text{lower\_bound} - the lower bound of the variable (default: 0)
- \text{upper\_bound} - the upper bound of the variable (default: None)
- \text{binary} - True if the variable is binary (default: False).
- \text{continuous} - True if the variable is continuous (default: True).
- \text{integer} - True if the variable is integral (default: False).
- \text{obj} - (optional) coefficient of this variable in the objective function (default: 0)
- \text{name} - an optional name for the newly added variable (default: None).
- \text{coefficients} – (optional) an iterable of pairs \((i, v)\). In each pair, \(i\) is a variable index (integer) and \(v\) is a value (element of \text{base\_ring()}).
OUTPUT: The index of the newly created variable

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.nrows()
0
sage: p.add_variable()
0
sage: p.nrows()
1
sage: p.add_variable(continuous=True, integer=True)
Traceback (most recent call last):
  ... ValueError: ...
sage: p.add_variable(name='x', obj=1)
1
sage: p.col_name(1)
'x'
sage: p.objective_coefficient(1)
1
```

`base_ring()`

Return the base ring.

OUTPUT:

A ring. The coefficients that the chosen solver supports.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.base_ring()
Rational Field
```

`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.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.add_variable(lower_bound=None)
0
sage: p.col_bounds(0)
(None, None)
sage: p.variable_lower_bound(0, 0)
sage: p.col_bounds(0)
(0, None)
```
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.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.add_variable(name="I_am_a_variable")
0
sage: p.col_name(0)
'I_am_a_variable'
```

dictionary()

Return a dictionary representing the current basis.

EXAMPLES:

```
sage: p = MixedIntegerLinearProgram(maximization=True,
solver="InteractiveLP")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(11/2 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method...
where
sage: d = b.dictionary(); d
LP problem dictionary ...
set(d.basic_variables())
\{x1, x3\}
sage: d.basic_solution()
(17/8, 0)
```

get_objective_value()

Return the value of the objective function.

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

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.add_variables(2)
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()
(continues on next page)
```
get_variable_value (variable)

Return the value of a variable given by the solver.

**Note:** Behavior is undefined unless solve has been called before.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")

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
```

**interactive_lp_problem()**

Return the *InteractiveLPProblem* object associated with this backend.

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(maximization=True, solver="InteractiveLP")

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

sage: p.add_constraint(-x[0] + x[1] <= 2)

sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)

sage: p.set_objective(11/2 * x[0] - 3 * x[1])

sage: b = p.get_backend()

sage: b.interactive_lp_problem()

LP problem ...
```

**is_maximization()**

Test whether the problem is a maximization.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver

sage: p = get_solver(solver = "InteractiveLP")

sage: p.is_maximization()

True

sage: p.set_sense(-1)

sage: p.is_maximization()

False
```
**is_slack_variable_basic**(*index*)

Test whether the slack variable of the given row is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

**INPUT:**

- `index` (integer) – the variable's id

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(maximization=True, 
                                  solver="InteractiveLP")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(11/2 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method
     # here
sage: b.solve() 0
sage: b.is_slack_variable_basic(0)
True
sage: b.is_slack_variable_basic(1)
False
```

**is_slack_variable_nonbasic_at_lower_bound**(*index*)

Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

**INPUT:**

- `index` (integer) – the variable's id

**EXAMPLES:**

```python
sage: p = MixedIntegerLinearProgram(maximization=True, 
                                  solver="InteractiveLP")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(11/2 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method
     # here
sage: b.solve() 0
sage: b.is_slack_variable_nonbasic_at_lower_bound(0)
False
sage: b.is_slack_variable_nonbasic_at_lower_bound(1)
True
```

**is_variable_basic**(*index*)

Test whether the given variable is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

**11.2. InteractiveLP Backend**
is_variable_binary(index)
Test whether the given variable is of binary type.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
get_solver
sage: p = get_solver(solver = "InteractiveLP")
p = MixedIntegerLinearProgram(solver="InteractiveLP")
sage: x = p.new_variable(nonnegative=True)
p.add_constraint(-x[0] + x[1] <= 2)
p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
p.set_objective(11/2 * x[0] - 3 * x[1])
b = p.get_backend()
# Backend-specific commands to instruct solver to use simplex method...
...here
sage: b.solve()
0
sage: b.is_variable_basic(0)
True
sage: b.is_variable_basic(1)
False
```

is_variable_continuous(index)
Test whether the given variable is of continuous/real type.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
get_solver
sage: p = get_solver(solver = "InteractiveLP")
p = MixedIntegerLinearProgram(solver="InteractiveLP")
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

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
get_solver
sage: p = get_solver(solver = "InteractiveLP")
p = MixedIntegerLinearProgram(solver="InteractiveLP")
p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_integer(0)
True
```
INPUT:

- index (integer) – the variable's id

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_integer(0)
False
```

is_variable_nonbasic_at_lower_bound(index)

Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

- index (integer) – the variable's id

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(maximization=True, solver="InteractiveLP")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(11/2 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: # Backend-specific commands to instruct solver to use simplex method here
sage: b.solve()
0
sage: b.is_variable_nonbasic_at_lower_bound(0)
False
sage: b.is_variable_nonbasic_at_lower_bound(1)
True
```

ncols()

Return the number of columns/variables.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLES:
from sage.numerical.backends.generic_backend import get_solver

p = get_solver(solver = "InteractiveLP")
print(p.nrows())
p.add_linear_constraints(2, 0, None)
print(p.nrows())

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

EXAMPLES:

sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.add_variable()
0
sage: p.objective_coefficient(0)
0
sage: p.objective_coefficient(0,2)
2

objective_constant_term (d=None)
Set or get the constant term in the objective function

INPUT:

• d (double) – its coefficient. If None (default), return the current value.

EXAMPLES:

sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.objective_constant_term()
0
sage: p.objective_constant_term(42)
42

problem_name (name=None)
Return or define the problem’s name

INPUT:

• name (str) – the problem’s name. When set to None (default), the method returns the problem’s name.

EXAMPLES:

sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.problem_name("There_once_was_a_french_fry")
remove_constraint \( (i) \)
Remove a constraint.

**INPUT:**

- \( i \) – index of the constraint to remove.

**EXAMPLES:**

```sage
sage: p = MixedIntegerLinearProgram(solver="InteractiveLP")
sage: v = p.new_variable(nonnegative=True)
sage: x, y = v[0], v[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()
47/5
sage: p.remove_constraint(0)
sage: p.solve()
10
sage: p.get_values([x,y])
[0, 3]
```

row \( (i) \)
Return a row

**INPUT:**

- index (integer) – the constraint’s id.

**OUTPUT:**

A pair \((\text{indices}, \text{coeffs})\) where \text{indices} lists the entries whose coefficient is nonzero, and to which \text{coeffs} associates their coefficient on the model of the add_linear_constraint method.

**EXAMPLES:**

```sage
sage: from sage.numerical.backends.generic_backend import get_solver
data: p = get_solver(solver = "InteractiveLP")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(zip(range(5), range(5)), 0, None)
sage: p.row(0)
([1, 2, 3, 4], [1, 2, 3, 4])
```

row_bounds \( (\text{index}) \)
Return the bounds of a specific constraint.

**INPUT:**

- index (integer) – the constraint’s id.

**OUTPUT:**

A pair \((\text{lower_bound}, \text{upper_bound})\). Each of them can be set to None if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

**EXAMPLES:**

```sage

```
\texttt{sage: from sage.numerical.backends.generic_backend import get\_solver}
\texttt{sage: p = get\_solver(solver = "InteractiveLP")}
\texttt{sage: p.add\_variables(5)}
\texttt{4}
\texttt{sage: p.add\_linear\_constraint(zip(range(5), range(5)), 2, 2)}
\texttt{sage: p.row\_bounds(0)}
\texttt{(2, 2)}

\textbf{row\_name (index)}

Return the \texttt{index} th row name.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{index} (integer) – the row’s id
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: from sage.numerical.backends.generic_backend import get\_solver}
\texttt{sage: p = get\_solver(solver = "InteractiveLP")}
\texttt{sage: p.add\_linear\_constraints(1, 2, None, names=['Empty constraint 1'])}
\texttt{sage: p.row\_name(0)}
\texttt{'Empty constraint 1'}
\end{verbatim}

\textbf{set\_objective (coeff, d=0)}

Set the objective function.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{coeff} – a list of real values, whose \texttt{i}-th element is the coefficient of the \texttt{i}-th variable in the objective function.
  \item \texttt{d} (real) – the constant term in the linear function (set to 0 by default)
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: from sage.numerical.backends.generic_backend import get\_solver}
\texttt{sage: p = get\_solver(solver = "InteractiveLP")}
\texttt{sage: p.add\_variables(5)}
\texttt{4}
\texttt{sage: p.set\_objective([1, 1, 2, 1, 3])}
\texttt{[p.objective\_coefficient(x) for x in range(5)]}
\texttt{[1, 1, 2, 1, 3]}
\end{verbatim}

Constants in the objective function are respected:

\begin{verbatim}
\texttt{sage: p = MixedIntegerLinearProgram(solver='InteractiveLP')}
\texttt{sage: x, y = p[0], p[1]}
\texttt{sage: p.add\_constraint(2\*x + 3\*y, max = 6)}
\texttt{sage: p.add\_constraint(3\*x + 2\*y, max = 6)}
\texttt{sage: p.set\_objective(x + y + 7)}
\texttt{p.solve()}
\texttt{47/5}
\end{verbatim}

\textbf{set\_sense (sense)}

Set the direction (maximization/minimization).

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{sense} (integer):
    \begin{itemize}
      \item \texttt{+1} => Maximization
    \end{itemize}
\end{itemize}
- -1 => Minimization

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
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.

In this backend, variables are always continuous (real). If integer or binary variables are requested via the parameter `vtype`, an error will be raised.

**INPUT:**

- *variable* (integer) – the variable’s id
- *vtype* (integer):
  - 1 Integer
  - 0 Binary
  - -1 Continuous

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,-1)
```

**set_verbosity** (*level*)

Set the log (verbosity) level

**INPUT:**

- *level* (integer) – From 0 (no verbosity) to 3.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "InteractiveLP")
```

**solve()**

Solve the problem.

**Note:** This method raises `MIPSolverException` exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)
EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
game: p = get_solver(solver = "InteractiveLP")
ensemble: p.add_linear_constraints(5, 0, None)
game: p.add_col(list(range(5)), list(range(5)))
game: p.solve()
0
sage: p.objective_coefficient(0,1)
game: 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 no lower bound. When set to False (default), the method returns the current value.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
game: p = get_solver(solver = "InteractiveLP")
game: p.add_variable(lower_bound=None)
0
sage: p.col_bounds(0)
(0, None)
sage: p.variable_lower_bound(0)
is None
True
sage: p.variable_lower_bound(0, 0)
sage: p.col_bounds(0)
(0, None)
sage: p.variable_lower_bound(0)
0
sage: p.variable_lower_bound(0, None)
sage: p.variable_lower_bound(0) is None
True
```

`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 no upper bound. When set to False (default), the method returns the current value.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
game: p = get_solver(solver = "InteractiveLP")
game: p.add_variable(lower_bound=None)
0
sage: p.col_bounds(0)
(continues on next page)```
11.3 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: GenericBackend

MIP Backend that uses the GLPK solver.

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 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.

EXAMPLES:

```python
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)

**EXAMPLES:**

```
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])
```

**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)

**EXAMPLES:**

```
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)
(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_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 continuous (default: True).
• integer - True if the variable is integral (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

EXAMPLES:

```python
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 their 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.

EXAMPLES:

```python
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, obj=42.0, 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.

EXAMPLES:

```python
sage: # needs sage.graphs
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))
....:     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.

EXAMPLES:
col_name(index)

Return the index th col name

INPUT:

• index (integer) – the col's id

EXAMPLES:

```
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')
sage: p.col_name(0)
'I am a variable'
```

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 structural 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 and the variable with index $k$ must not be basic. Otherwise, a ValueError is 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.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
sage: lp.add_variables(3)
2
```
Numerical Optimization, Release 10.3

(continued from previous page)

```python
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
sage: lp.add_linear_constraint(list(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):
... 
ValueError: basis factorization does not exist
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):
... 
ValueError: slack variable 0 is basic
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 structural 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 and the variable with index $k$ must be basic. Otherwise, a `ValueError` is 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.

**EXAMPLES:**
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")

```python
sage: lp.add_variables(3)
```

2

```python
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
```

```python
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
```

```python
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [2, 1.5, 0.5])), None, 8)
```

```python
sage: lp.set_objective([60, 30, 20])
```

```python
sage: import sage.numerical.backends.glpk_backend as backend
```

```python
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
```

```python
sage: lp.eval_tab_row(0)
```

Traceback (most recent call last):
...

```
ValueError: basis factorization does not exist
```

```python
sage: lp.solve()
```

0

```python
sage: lp.eval_tab_row(0)
```

```python
([[1, 2, 4], [-2.0, 8.0, -2.0]])
```

```python
sage: lp.eval_tab_row(3)
```

```python
([[1, 2, 4], [-0.5, 1.5, -1.25]])
```

```python
sage: lp.eval_tab_row(5)
```

```python
([[1, 2, 4], [2.0, -4.0, 2.0]])
```

```python
sage: lp.eval_tab_row(-1)
```

Traceback (most recent call last):
...

```
ValueError: slack variable 1 is not basic
```

```python
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.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
```

```python
sage: p.add_variables(3)
```

2

```python
sage: p.add_linear_constraint(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
```

```python
sage: p.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
```

```python
sage: p.add_linear_constraint(list(zip([0, 1, 2], [2, 1.5, 0.5])), None, 8)
```

```python
sage: p.set_objective([60, 30, 20])
```

```python
sage: import sage.numerical.backends.glpk_backend as backend
```

(continues on next page)
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

**EXAMPLES:**

```python
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(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
sage: lp.add_linear_constraint(list(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(100)
Traceback (most recent call last):
  ... ValueError: The variable's index j must satisfy 0 <= j < number_of_variables
```

get_objective_value()

Returns the value of the objective function.

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

**EXAMPLES:**

```python
```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
1
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
7.5
sage: p.set_objective([2, 5])
1.5
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 \((\text{bestinteger} - \text{bestobjective})/(1e^{-10+|\text{bestobjective}|})\), where \text{bestinteger} is the value returned by \text{get_objective_value()} and \text{bestobjective} is the value returned by \text{best_known_objective_bound()}. For a maximization problem, the value is computed by \(((\text{bestobjective} - \text{bestinteger})/(1e^{-10 + |\text{bestobjective}|})).\n
Note: Has no meaning unless solve has been called before.

EXAMPLES:

sage: # needs sage.graphs
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

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:

- \text{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.

**EXAMPLES:**

```python
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(list(zip([0, 1, 2], [8, 6, 1])), None, 48)

sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)

sage: lp.add_linear_constraint(list(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** 
Returns the value of the auxiliary variable associated with i-th row.

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

**EXAMPLES:**

```python
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(list(zip([0, 1, 2], [8, 6, 1])), None, 48)

sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)

sage: lp.add_linear_constraint(list(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** 
Retrieves 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

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")
```

```
sage: lp.add_variables(3)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
sage: lp.add_linear_constraint(list(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()
sage: lp.get_row_stat(0)
sage: lp.get_row_stat(1)
sage: lp.get_row_stat(-1)
```

```
ValueError: The constraint's index \( i \) must satisfy \( 0 <= i < \) number_of_constraints
```

```
get_variable_value(variable)

Returns the value of a variable given by the solver.

Note: Behaviour is undefined unless \texttt{solve} has been called before.
```

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
sage: p.add_linear_constraint([[0, 1], [1, 2]], None, 3)
sage: p.set_objective([2, 5])
sage: p.solve()
sage: p.get_objective_value()
```

(continues on next page)
**is_maximization()**

Test whether the problem is a maximization

EXAMPLES:

```python
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_slack_variable_basic(index)**

Test whether the slack variable of the given row is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

- `index` (integer) – the variable's id

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(maximization=True, solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: import sage.numerical.backends.glpk_backend as backend
sage: b.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: b.solve()
0
sage: b.is_slack_variable_basic(0)
True
sage: b.is_slack_variable_basic(1)
False
```

**is_slack_variable_nonbasic_at_lower_bound(index)**

Test whether the slack variable of the given row is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

- `index` (integer) – the variable's id

EXAMPLES:
```
sage: p = MixedIntegerLinearProgram(maximization=True,  
                          solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: import sage.numerical.backends.glpk_backend as backend
sage: b.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: b.solve()
0
sage: b.is_slack_variable_nonbasic_at_lower_bound(0)
False
sage: b.is_slack_variable_nonbasic_at_lower_bound(1)
True
```

**is_variable_basic** (*index*)

Test whether the given variable is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

**INPUT:**

- *index* (integer) – the variable’s id

**EXAMPLES:**

```
sage: p = MixedIntegerLinearProgram(maximization=True,  
                          solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: import sage.numerical.backends.glpk_backend as backend
sage: b.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: b.solve()
0
sage: b.is_variable_basic(0)
True
sage: b.is_variable_basic(1)
False
```

**is_variable_binary** (*index*)

Test whether the given variable is of binary type.

**INPUT:**

- *index* (integer) – the variable’s id

**EXAMPLES:**

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.ncols()
0
```

(continues on next page)
is_variable_continuous(index)

Test whether the given variable is of continuous/real type.

INPUT:

- index (integer) – the variable's id

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
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

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
0
sage: p.add_variable()
0
sage: p.is_variable_integer(0)
True
```

is_variable_nonbasic_at_lower_bound(index)

Test whether the given variable is nonbasic at lower bound. This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

- index (integer) – the variable’s id

EXAMPLES:

```python
sage: p = MixedIntegerLinearProgram(maximization=True,
solver="GLPK")
sage: x = p.new_variable(nonnegative=True)
(...)
```
\begin{verbatim}
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
\end{verbatim}

**ncols()**

Return the number of columns/variables.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

**nrows()**

Return the number of rows/constraints.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

**objective_coefficient** *(variable, coeff=None)*

Set or get the coefficient of a variable in the objective function.

**INPUT:**

- \texttt{variable} (integer) – the variable’s id
- \texttt{coeff} (double) – its coefficient or \texttt{None} for reading (default: \texttt{None})

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}
print_ranges (filename=None)

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.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
sage: p.add_variables(2)
sage: p.add_linear_constraint(list(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: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(mode="r+t", suffix=".tmp") as f:
    .....: p.print_ranges(f.name)
    .....: for ll in f.readlines():
    .....:     if ll: print(ll)
...
GLPK ... - SENSITIVITY ANALYSIS REPORT

Problem:
Objective: 7.5 (MAXimum)

<table>
<thead>
<tr>
<th>No.</th>
<th>Row name</th>
<th>St</th>
<th>Activity</th>
<th>Slack</th>
<th>Lower bound</th>
<th>Marginal</th>
<th>Upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NU</td>
<td></td>
<td>3.00000</td>
<td>.</td>
<td>-Inf</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2.50000</td>
<td>.</td>
<td>2.50000</td>
<td>3.00000</td>
<td></td>
</tr>
</tbody>
</table>
### problem_name(name=None)

Return or define the problem's name

**INPUT:**

- name (str) – the problem's name. When set to None (default), the method returns the problem's name.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
tsage: p = get_solver(solver = "GLPK")
tsage: p.problem_name("There once was a french fry")
tsage: 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

**EXAMPLES:**

```python
tsage: p = MixedIntegerLinearProgram(solver='GLPK')
tsage: x, y = p['x'], p['y']
tsage: p.add_constraint(2*x + 3*y <= 6)
tsage: p.add_constraint(3*x + 2*y <= 6)
tsage: p.add_constraint(x >= 0)
tsage: p.set_objective(x + y + 7)
tsage: p.solve()  # 9.0
ntsage: p.remove_constraint(0)
tsage: p.solve()  # 10.0
```
Removing fancy constraints does not make Sage crash:

```python
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.

**EXAMPLES:**

```python
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]
```

**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.

**EXAMPLES:**

```python
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(list(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.

EXAMPLES:

```python
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(list(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 indexth row name

INPUT:

• index (integer) – the row’s id

EXAMPLES:

```python
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

EXAMPLES:

```python
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(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
sage: lp.add_linear_constraint(list(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)
(continues on next page)"
**set_objective** \((\text{coeff}, d=0.0)\)

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)

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")

sage: p.add_variables(5)

sage: p.set_objective([1, 1, 2, 1, 3])

sage: [p.objective_coefficient(x) for x in range(5)]
[1.0, 1.0, 2.0, 1.0, 3.0]
```

**set_row_stat** \((i, \text{stat})\)

Set the status of a constraint.

**INPUT:**

- **\(i\)** – The index of the constraint
- **\(\text{stat}\)** – The status to set to

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: lp = get_solver(solver = "GLPK")

sage: lp.add_variables(3)

sage: lp.add_linear_constraint(list(zip([0, 1, 2], [8, 6, 1])), None, 48)

sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)

sage: lp.add_linear_constraint(list(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** \((\text{sense})\)

Set the direction (maximization/minimization).

**INPUT:**

- **\(\text{sense}\)** (integer):

```python

```
- +1 => Maximization
- -1 => Minimization

EXAMPLES:

```python
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

EXAMPLES:

```python
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.

EXAMPLES:

```python
sage: p.<x> = MixedIntegerLinearProgram(solver="GLPK")
sage: p.add_constraint(10 * x[0] <= 1)
sage: p.add_constraint(5 * x[1] <= 1)
sage: p.set_objective(x[0] + x[1])
sage: p.solve()
0.30000000000000004
sage: p.get_backend().set_verbosity(3)
sage: p.solver_parameter("simplex_or_intopt", "intopt_only")
sage: p.solve()
GLPK Integer Optimizer...
2 rows, 2 columns, 2 non-zeros
0 integer variables, none of which are binary
```

(continues on next page)
Preprocessing...
Objective value = 3.000000000e-01
INTEGER OPTIMAL SOLUTION FOUND BY MIP PREPROCESSOR
0.30000000000000004

```
sage: p.<x> = MixedIntegerLinearProgram(solver="GLPK/exact")
sage: p.add_constraint(10 * x[0] <= 1)
sage: p.add_constraint(5 * x[1] <= 1)
sage: p.set_objective(x[0] + x[1])
sage: p.solve() # tol 1e-14
0.3
sage: p.get_backend().set_verbosity(2)
sage: p.solve() # tol 1e-14
* 2: objval = 0.3 (0)
* 2: objval = 0.3 (0)
0.3
sage: p.get_backend().set_verbosity(3)
sage: p.solve() # tol 1e-14
glp_exact: 2 rows, 2 columns, 2 non-zeros
... * 2: objval = 0.3 (0)
* 2: objval = 0.3 (0)
OPTIMAL SOLUTION FOUND
0.3
```

`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". By default, the simplex method will be used first to detect pathological problems that the integer solver cannot handle. If all variables are continuous, the integer algorithm reduces to solving the linear program by the simplex method.

EXAMPLES:

```
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 cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

EXAMPLES:
```python
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:** GLPK's `glp_intopt` sometimes fails catastrophically when given a system it cannot solve (github issue #12309). It can loop indefinitely, or just plain segfault. Upstream considers this behavior “essentially innate” to the current design, and suggests preprocessing with `glp_simplex`, which is what SageMath does by default. Set the `simplex_or_intopt` solver parameter to `glp_intopt_only` at your own risk.

### EXAMPLES:

```python
sage: lp = MixedIntegerLinearProgram(solver = "GLPK")

sage: v = lp.new_variable(nonnegative=True)


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):
...
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.

### EXAMPLES:

```python
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`, `get_dual` etc.) or tableau data (`get_r`, `get_t`, `get_s` 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, which means that rationals in the input data may be rounded before the exact solver sees them. Thus, it is unreasonable to expect that arbitrary LPs with rational coefficients are solved exactly. Once the LP has been read into the backend, it reconstructs rationals from doubles and does solve exactly over the rationals, but results are returned as doubles.

**EXAMPLES:**

```
sage: lp.solver_parameter(“simplex_or_intopt”, “exact_simplex_only”) # use...
˓→exact simplex only
sage: lp.solve()
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.

**EXAMPLES:**

```
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.

**EXAMPLES:**

```
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 only
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
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 (github issue #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
```

(continues on next page)
MIPSolverException: GLPK: Problem has unbounded solution

```python
sage: lp.solver_parameter("simplex_or_intopt", "simplex_then_intopt")
sage: lp.solve()
```

Traceback (most recent call last):

```python
MIPSolverException: GLPK: The LP (relaxation) problem has no dual feasible solution
```

```python
sage: lp.solver_parameter("simplex_or_intopt", "intopt_only")
sage: lp.solve()
```

Traceback (most recent call last):

```python
MIPSolverException: GLPK: The LP (relaxation) problem has no dual feasible solution
```

```python
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:

```python
sage: # needs sage.graphs
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:

```python
sage: # needs sage.graphs
sage: p.solver_parameter("mip_gap_tolerance",1)
sage: p.solver_parameter("timelimit",3.0)
sage: p.solve() # rel tol 100
```

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/](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 specified 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:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timelimit</td>
<td>specify the time limit IN SECONDS. This affects both simplex and intopt.</td>
</tr>
<tr>
<td>timelimit_simplex</td>
<td>specify the time limit IN MILLISECONDS. (This is glpk’s default.)</td>
</tr>
<tr>
<td>simplex_or_intopt</td>
<td>specify which solution routines in GLPK to use. Set this to either simplex_only, exact_simplex_only, intopt_only, or simplex_then_intopt (the default). The simplex_then_intopt option does some extra work, but avoids hangs/crashes in GLPK on problems with no solution; SageMath will try simplex first, then perform integer optimization only if a solution of the LP relaxation exists. If you know that your system is not pathological, one of the other options will be faster.</td>
</tr>
<tr>
<td>verbosity_intopt and</td>
<td>one of GLP_MSG_OFF, GLP_MSG_ERR, GLP_MSG_ON, or GLP_MSG_ALL. The default is GLP_MSG_OFF.</td>
</tr>
<tr>
<td>verbosity_simplex</td>
<td>the output frequency, in milliseconds. Default is 5000.</td>
</tr>
<tr>
<td>output_frequency_intopt</td>
<td>the output frequency, in milliseconds. Default is 5000.</td>
</tr>
<tr>
<td>and output_frequency_simplex</td>
<td></td>
</tr>
<tr>
<td>output_delay_intopt and</td>
<td>the output delay, in milliseconds, regarding the use of the simplex method on the LP relaxation. Default is 10000.</td>
</tr>
<tr>
<td>output_delay_simplex</td>
<td></td>
</tr>
</tbody>
</table>

**intopt-specific parameters:**
<table>
<thead>
<tr>
<th><strong>branching</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• GLP_BR_FFV first fractional variable</td>
<td></td>
</tr>
<tr>
<td>• GLP_BR_LFV last fractional variable</td>
<td></td>
</tr>
<tr>
<td>• GLP_BR_MFV most fractional variable</td>
<td></td>
</tr>
<tr>
<td>• GLP_BR_DTH Driebeck-Tomlin heuristic (default)</td>
<td></td>
</tr>
<tr>
<td>• GLP_BR_PCH hybrid pseudocost heuristic</td>
<td></td>
</tr>
<tr>
<td><strong>backtracking</strong></td>
<td></td>
</tr>
<tr>
<td>• GLP_BT_DFS depth first search</td>
<td></td>
</tr>
<tr>
<td>• GLP_BT_BFS breadth first search</td>
<td></td>
</tr>
<tr>
<td>• GLP_BT_BLB best local bound (default)</td>
<td></td>
</tr>
<tr>
<td>• GLP_BT_BPH best projection heuristic</td>
<td></td>
</tr>
<tr>
<td><strong>preprocessing</strong></td>
<td></td>
</tr>
<tr>
<td>• GLP_PP_NONE</td>
<td></td>
</tr>
<tr>
<td>• GLP_PP_ROOT preprocessing only at root level</td>
<td></td>
</tr>
<tr>
<td>• GLP_PP_ALL (default)</td>
<td></td>
</tr>
<tr>
<td><strong>feasibility_pump</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
<tr>
<td><strong>gomory_cuts</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
<tr>
<td><strong>mixed_int_rounding_cuts</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
<tr>
<td><strong>mixed_cover_cuts</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
<tr>
<td><strong>clique_cuts</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
<tr>
<td><strong>absolute_tolerance</strong></td>
<td>(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.”</td>
</tr>
<tr>
<td><strong>relative_tolerance</strong></td>
<td>(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.”</td>
</tr>
<tr>
<td><strong>mip_gap_tolerance</strong></td>
<td>(double) relative mip gap tolerance. Default is 0.0.</td>
</tr>
<tr>
<td><strong>presolve_intopt</strong></td>
<td>GLP_ON (default) or GLP_OFF.</td>
</tr>
<tr>
<td><strong>binarize</strong></td>
<td>GLP_ON or GLP_OFF (default)</td>
</tr>
</tbody>
</table>

**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.

---

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
glpk
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`

```python
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.

**EXAMPLES:**
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:

```python
sage: import sage.numerical.backends.glpk_backend as backend
sage: lp.solver_parameter(backend.glp_simplex_or_intopt, backend.glp_simplex_only)
sage: lp.solver_parameter(backend.glp_mir_cuts, backend.glp_on)
```

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.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK")
```

**variable_upper_bound** *(index, value=False)*

Return or define the upper bound on a variable

**INPUT:**

- **index** (integer) – the variable’s id
Numerical Optimization, Release 10.3

- 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.

EXAMPLES:

```python
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)
```

`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.

EXAMPLES:

```python
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(list(zip([0, 1, 2], [8, 6, 1])), None, 48)
sage: lp.add_linear_constraint(list(zip([0, 1, 2], [4, 2, 1.5])), None, 20)
sage: lp.add_linear_constraint(list(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)

EXAMPLES:
write_mps (filename, modern)
Write the problem to a .mps file

INPUT:

• filename (string)

EXAMPLES:

11.4 GLPK/Exact Backend (simplex method in exact rational arithmetic)

AUTHORS:

• Matthias Koeppe (2016-03)

class sage.numerical.backends.glpk_exact_backend.GLPKExactBackend
Bases: GLPKBackend

MIP Backend that runs the GLPK solver in exact rational simplex mode.

The only access to data is via double-precision floats, which means that rationals in the input data may be rounded before the exact solver sees them. Thus, it is unreasonable to expect that arbitrary LPs with rational coefficients are solved exactly. Once the LP has been read into the backend, it reconstructs rationals from doubles and does solve exactly over the rationals, but results are returned as as doubles.

There is no support for integer variables.
**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 nonnegative and real.

In this backend, variables are always continuous (real). If integer variables are requested via the parameters *binary* and *integer*, an error will be raised.

**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

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
go
sage: p = get_solver(solver = "GLPK/exact")
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.objective_coefficient(4)
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 variables.

This amounts to adding new columns to the matrix. By default, the variables are both nonnegative and real.

In this backend, variables are always continuous (real). If integer variables are requested via the parameters *binary* and *integer*, an error will be raised.

**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.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK/exact")
sage: p.ncols()
0
sage: p.add_variables(5)
4
sage: p.ncols()
5
sage: p.add_variables(2, lower_bound=-2.0, obj=42.0, names=['a','b'])
6
```

**set_variable_type** *(variable, vtype)*

Set the type of a variable.

In this backend, variables are always continuous (real). If integer or binary variables are requested via the parameter *vtype*, an error will be raised.

INPUT:

- **variable** (integer) – the variable’s id
- **vtype** (integer):
  - 1 Integer
  - 0 Binary
  - -1 Real

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "GLPK/exact")
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: ...
```
11.5 GLPK Backend for access to GLPK graph functions

AUTHORS:

- Christian Kuper (2012-11): Initial implementation

11.5.1 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 dictionaries 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.

11.5.2 Classes and methods

```python
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:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
</table>
| plain  | Read data from a plain text file containing the following information:  
  nv na  
  i[1] j[1]  
  ...  
  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 description of the DIMACS format can be found at [http://dimacs.rutgers.edu/Challenges/](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: import tempfile  
sage: with tempfile.NamedTemporaryFile() as f:  
    ....:     _ = gbe.write_graph(f.name)  
    ....:     gbe1 = GLPKGraphBackend(f.name, "plain")
```

Writing graph to ...  
4 lines were written  
Reading graph from ...
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(sort=False):
....:   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

**EXAMPLES:**

```
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 iterable container of triples of the form *(u, v, params)* where params is a dict as described in `add_edge`.

**EXAMPLES:**

```
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()

Note: The code snippet has been modified to fit the context.

add_vertex (name=None)
Adds an isolated vertex to the graph.

If the vertex already exists, nothing is done.

INPUT:

- name – str 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.

EXAMPLES:

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.

EXAMPLES:
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]

cpp()
Solves the critical path problem of a project network.

OUTPUT:
The length of the critical path of the network

EXAMPLES:

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('i')
Sage: i, 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 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()

EXAMPLES:

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}))
(continues on next page)
sage: edges.append(("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":20.0, "cost":20}))
sage: gbe.add_edges(edges)

delete_edges(edges)

Deletes edges from the graph.
Non existing edges are ignored.

INPUT:

• edges – An iterable container of edges.

See also:

delete_edge()

EXAMPLES:

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":20.0, "cost":20}))
sage: gbe.add_edges(edges)

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.

EXAMPLES:

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

**EXAMPLES:**

```python
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.

**EXAMPLES:**

```python
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

EXAMPLES:

```python
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: 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)

EXAMPLES:

```python
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 dictionaries associated to each vertex.

INPUT:

• verts – iterable container of vertices

OUTPUT:
A list of pairs \((\text{vertex}, \text{ properties})\) where \text{properties} is a dictionary containing the numerical values associated with a vertex. For more information, see the documentation of \texttt{GLPKGraphBackend.get_vertex}.

**EXAMPLES:**

```python
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())
[(\text{cut'}, 0), (\text{es'}, 0.0), (\text{ls'}, 0.0), (\text{pi'}, 0.0), (\text{rhs'}, 0.0)]
sage: gbe.get_vertices(['C', 'D'])
{}
```

**maxflow_ffalg** \((u=\text{None}, v=\text{None})\)

Finds solution to the maxflow problem with Ford-Fulkerson algorithm.

**INPUT:**

- \(u\) – Name (as \texttt{str}) of the tail vertex. Default is \texttt{None}.
- \(v\) – Name (as \texttt{str}) of the head vertex. Default is \texttt{None}.

If \(u\) or \(v\) are \texttt{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 \texttt{IndexError} is raised.
- If the source and sink are identical, a \texttt{ValueError} is raised.
- This method raises \texttt{MIPSolverException} exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

**EXAMPLES:**

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: v = gbe.add_vertices([\texttt{None} for \texttt{i} in \texttt{range(5)}])
sage: edges = ((0, 1, 2), (0, 2, 3), (1, 2, 3), (1, 3, 4),
...... (3, 4, 1), (2, 4, 2))
sage: for \texttt{a} in edges:
...... edge = gbe.add_edge(str(a[0]), str(a[1]), \{'\text{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):
...
\texttt{IndexError}: Source or sink vertex does not exist
```
Numerical Optimization, Release 10.3

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 cannot be computed for
any reason (none exists, or the LP solver was not able to find it, etc…)
EXAMPLES:
sage:
sage:
sage:
sage:
sage:
sage:
....:
sage:
sage:

from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
gbe = GLPKGraphBackend()
vertices = (35, 50, 40, -45, -20, -30, -30)
vs = gbe.add_vertices([None for i in range(len(vertices))])
v_dict = {}
for i, v in enumerate(vs):
v_dict[v] = vertices[i]
gbe.set_vertices_demand(list(v_dict.items()))
cost = ((8, 6, 10, 9), (9, 12, 13, 7), (14, 9, 16, 5))

sage: for i in range(len(cost)):
....:
for j in range(len(cost[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("{} -> {} {}".format(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 algorithm (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.
EXAMPLES:

11.5. GLPK Backend for access to GLPK graph functions

217


```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
```

```python
vertices = [None for i in range(3)]
```

```python
gbe.add_vertices(vertices)
```

```python
['0', '1', '2']
```

```python
sage: gbe.set_vertex_demand('0', 2)
```

```python
gbe.get_vertex('0')['rhs']
```

```python
2.0
```

```python
gbe.set_vertex_demand('3', 2)
```

```python
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()`.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
```

```python
vertices = [None for i in range(3)]
```

```python
gbe.add_vertices(vertices)
```

```python
['0', '1', '2']
```

```python
gbe.set_vertices_demand([(0, 2), (1, 3), (3, 4)])
```

```python
sage: sorted(gbe.get_vertex('1').items())
```

```python
[(‘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.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
```

```python
verts = ['A', 'B', 'C']
```

```python
gbe.add_vertices(verts)
```

```python
gbe.vertices()
```

```python
['A', 'B', 'C']
```

```python
gbe.vertices()
```

```python
['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 description 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

EXAMPLES:

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: a = gbe.add_edge("0", "1")
```

```python
sage: with tempfile.NamedTemporaryFile() as f:
    ....:
    gbe.write_ccdata(f.name)
```

```
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

EXAMPLES:

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: a = gbe.add_edge("0", "1")
```

```python
sage: with tempfile.NamedTemporaryFile() as f:
    ....:
    gbe.write_graph(f.name)
```

```
Writing graph to ...
4 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

EXAMPLES:

```python
sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
sage: gbe = GLPKGraphBackend()
sage: import tempfile
sage: with tempfile.NamedTemporaryFile() as f:
    ....:
```

(continues on next page)
....:  gbe.write_maxflow(f.name)
Traceback (most recent call last):
...
OSError: Cannot write empty graph

.. literal-block:: python

    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: with tempfile.NamedTemporaryFile() as f:
    ....:
    ....:  gbe.write_maxflow(f.name)
    Writing maximum flow problem data to ...
    6 lines were written 0

**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

**EXAMPLES:**

.. literal-block:: python

    sage: from sage.numerical.backends.glpk_graph_backend import GLPKGraphBackend
    sage: gbe = GLPKGraphBackend()
    sage: a = gbe.add_edge("0", "1")
    sage: import tempfile
    sage: with tempfile.NamedTemporaryFile() as f:
    ....:
    ....:  gbe.write_mincost(f.name)
    Writing min-cost flow problem data to ...
    4 lines were written 0

---

**11.6 PPL Backend**

**AUTHORS:**

- Risan (2012-02): initial implementation
- Jeroen Demeyer (2014-08-04) allow rational coefficients for constraints and objective function (github issue #16755)

**class** *sage.numerical.backends.ppl_backend.PPLBackend*

- Bases: *GenericBackend*

  MIP Backend that uses the exact MIP solver from the Parma Polyhedra Library.

**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 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.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
dsage: p = get_solver(solver = "PPL")
dsage: p.ncols()
0
dsage: p.nrows()
0
dsage: p.add_linear_constraints(5, 0, None)
dsage: p.add_col(zip(range(5)), list(range(5)))
dsage: 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)

EXAMPLES:

```
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)
```
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)

EXAMPLES:

```
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=False, 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 continuous (default: True).
• integer – True if the variable is integral (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

EXAMPLES:
add_variables \( (n, \text{lower\_bound}=0, \text{upper\_bound}=\text{None}, \text{binary}=\text{False}, \text{continuous}=\text{True}, \text{integer}=\text{False}, \text{obj}=0, \text{names}=\text{None}) \)

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)
- \( \text{lower\_bound} \) – the lower bound of the variable (default: 0)
- \( \text{upper\_bound} \) – the upper bound of the variable (default: None)
- \( \text{binary} \) – True if the variable is binary (default: False).
- \( \text{continuous} \) – True if the variable is continuous (default: True).
- \( \text{integer} \) – True if the variable is integral (default: False).
- \( \text{obj} \) – (optional) coefficient of all variables in the objective function (default: 0)
- \( \text{names} \) – optional list of names (default: None)

OUTPUT: The index of the variable created last.

EXAMPLES:
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.

EXAMPLES:

```
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.

EXAMPLES:

```
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
```

(continues on next page)
get_variable_value (variable)
Return the value of a variable given by the solver.

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

EXAMPLES:

```python
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.

EXAMPLES:

```python
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())
<class 'sage.rings.rational.Rational'>
sage: p.init_mip()
```

is_maximization()
Test whether the problem is a maximization

EXAMPLES:

```python
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

EXAMPLES:

```sage
from sage.numerical.backends.generic_backend import get_solver
p = get_solver(solver = "PPL")
p.ncols() 0
p.add_variable() 0
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

EXAMPLES:

```sage
from sage.numerical.backends.generic_backend import get_solver
p = get_solver(solver = "PPL")
p.ncols() 0
p.add_variable() 0
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

EXAMPLES:

```sage
from sage.numerical.backends.generic_backend import get_solver
p = get_solver(solver = "PPL")
p.ncols() 0
p.add_variable() 0
p.is_variable_integer(0)
False
```
**ncols()**

Return the number of columns/variables.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
0
sage: p.ncols()  # Initial number of columns
1
sage: p.add_variables(2)  # Add 2 new variables
1
sage: p.ncols()  # Total number of columns after adding variables
2
```

**nrows()**

Return the number of rows/constraints.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
0
sage: p.nrows()  # Initial number of rows
0
sage: p.add_linear_constraints(2, 2.0, None)  # Add 2 new linear constraints
sage: p.nrows()  # Total number of rows after adding constraints
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

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
0
sage: p.add_variable()  # Add a new variable
0
sage: p.objective_coefficient(0)  # Initial coefficient
0
sage: p.objective_coefficient(0, 2)  # Set coefficient to 2
sage: p.objective_coefficient(0)  # Coefficient after setting
2
```

**problem_name**(name=None)

Return or define the problem’s name

**INPUT:**

- name (str) – the problem’s name. When set to None (default), the method returns the problem’s name.

**EXAMPLES:**

```python
```
```
sage: from sage.numerical.backends.generic_backend import get_solver
g 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.

**EXAMPLES:**

```
sage: from sage.numerical.backends.generic_backend import get_solver
g sage: p = get_solver(solver = "PPL")
 sage: p.add_variables(5)
 4
g 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 row’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.

**EXAMPLES:**

```
sage: from sage.numerical.backends.generic_backend import get_solver
g sage: p = get_solver(solver = "PPL")
 sage: p.add_variables(5)
 4
g 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

**EXAMPLES:**
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:**

```python
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])
```

```python
sage: p.solve()
6/5
```

```python
sage: p.set_objective(x[0]/2 + 1)
```

```python
sage: p.show()
Maximization:
1/2 x_0 + 1
```

```python
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)
```

```python
sage: p.solve()
8/5
```

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
```

```python
sage: p.add_variables(5)
```

```python
sage: p.set_objective([1, 1, 2, 1, 3])
```

```python
sage: [p.objective_coefficient(x) for x in range(5)]
[1, 1, 2, 1, 3]
```

**set_sense** *(sense)*

Set the direction (maximization/minimization).

**INPUT:**

- sense (integer):
  - +1 => Maximization
  - -1 => Minimization

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
```

```python
sage: p.is_maximization()
```

(continues on next page)
True
sage: p.set_sense(-1)
sage: p.is_maximization()
False

set_variable_type(\texttt{variable}, \texttt{vtype})

Set the type of a variable.

INPUT:

\begin{itemize}
  \item \texttt{variable} (integer) – the variable’s id
  \item \texttt{vtype} (integer):
    \begin{itemize}
      \item 1 Integer
      \item 0 Binary
      \item -1 Continuous
    \end{itemize}
\end{itemize}

EXAMPLES:

sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver = "PPL")
sage: p.add_variables(4)
4
sage: p.set_variable_type(0, 1)
sage: p.is_variable_integer(0)
True
sage: p.set_variable_type(3, 0)
sage: p.is_variable_integer(3) \textbf{or} p.is_variable_binary(3)
True
sage: p.col_bounds(3) \# tol 1e-6
(0, 1)
sage: p.set_variable_type(3, -1)
sage: p.is_variable_continuous(3)
True

set_verbosity(\texttt{level})

Set the log (verbosity) level. Not Implemented.

EXAMPLES:

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 \texttt{MIPSolverException} exceptions when the solution cannot be computed for any reason (none exists, or the solver was not able to find it, etc…)

EXAMPLES:

A linear optimization problem:
An unbounded problem:

```python
sage: p.objective_coefficient(0,1)
sage: p.solve()
Traceback (most recent call last):
...  
MIPSolverException: ...
```

An integer optimization problem:

```python
sage: x = p.new_variable(integer=True, nonnegative=True)
sage: p.set_objective(x[0] + x[1] + 7)
sage: p.solve()
9
```

**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.

**EXAMPLES:**

```python
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 False (default), the method returns the current value.
EXAMPLES:

```python
class sage.numerical.backends.generic_backend.CVPBBackend:
    def add_col(self, 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 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.
```

11.7 CVXOPT Backend

AUTHORS:

- Ingolfur Edvardsson (2014-05): initial implementation

```python
class sage.numerical.backends.cvxopt_backend.CVXOPTBackend
    def add_col(self, indices, coeffs):
        # Add a column.
        self.add_variable()  # 0
        self.col_bounds(0)  # (0, None)
        self.variable_upper_bound(0, 5)
        self.col_bounds(0)  # (0, 5)
        self.variable_upper_bound(0, None)
        self.col_bounds(0)  # (0, None)
```

```python
class sage.numerical.backends.generic_backend.CVPBBackend:
    def add_col(self, 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 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.
```
**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)

**EXAMPLES:**

```python
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_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

**EXAMPLES:**
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

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.

EXAMPLES:

from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
(continues on next page)

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.

EXAMPLES:

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.

EXAMPLES:

```
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: N(p.get_objective_value(),4)
7.5
sage: N(p.get_variable_value(0),4)
3.6e-7
sage: N(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.

EXAMPLES:

```
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: N(p.get_objective_value(),4)
7.5
sage: N(p.get_variable_value(0),4)
3.6e-7
sage: N(p.get_variable_value(1),4)
1.5
```

is_maximization()

Test whether the problem is a maximization

EXAMPLES:
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

EXAMPLES:

```python
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

EXAMPLES:

```python
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

EXAMPLES:

```python
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)
```

```Python
Traceback (most recent call last):
... 
ValueError: ...
```

```
sage: p.is_variable_integer(0)
```

```
False
```

### ncols()

Return the number of columns/variables.

EXAMPLES:

```python
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.

EXAMPLES:

```python
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

EXAMPLES:
Numerical Optimization, Release 10.3

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
```

```python
sage: p.add_variable()
```

```python
sage: p.objective_coefficient(0)
```

```python
sage: p.objective_coefficient(0,2)
```

```python
sage: p.objective_coefficient(0)
```

```python
problem_name(name=None)
```

Return or define the problem's name

**INPUT:**

- name (str) – the problem's name. When set to `None` (default), the method returns the problem's name.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
```

```python
sage: p.problem_name("There once was a french fry")
```

```python
sage: print(p.problem_name())
```

```python
There once was a french fry
```

```python
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.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
```

```python
sage: p.add_variables(5)
```

```python
sage: p.add_linear_constraint(list(zip(range(5), range(5))), 2, 2)
```

```python
sage: p.row(0)
```

```python
([1, 2, 3, 4], [1, 2, 3, 4])
```

```python
sage: p.row_bounds(0)
```

```python
(2, 2)
```

```python
row_bounds(index)
```

Return the bounds of a specific constraint.

**INPUT:**

- index (integer) – the constraint's id.

**OUTPUT:**

```python
238 Chapter 11. Linear Optimization (LP) and Mixed Integer Linear Optimization (MIP) Solver backends
```
A pair \((\text{lower\_bound}, \text{upper\_bound})\). Each of them can be set to None if the constraint is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
generate: p = get_solver(solver="CVXOPT")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint(list(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

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
generate: 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)

EXAMPLES:

```
sage: from sage.numerical.backends.generic_backend import get_solver
generate: p = get_solver(solver="CVXOPT")
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
sage: [p.objective_coefficient(x) for x in range(5)]
[1, 1, 2, 1, 3]
```

`set_sense(sense)`

Set the direction (maximization/minimization).

INPUT:

- sense (integer):
  - +1 => Maximization
  - -1 => Minimization

EXAMPLES:
```python
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.

EXAMPLES:

```python
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 cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc…)

EXAMPLES:

```python
c
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: N(p.solve(), digits=2)
-9.0
```

```python
c
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: N(p.solve(), digits=4)
48.83
```

(continues on next page)
When the optimal solution is not unique, CVXOPT as an interior point solver gives a different type of solution compared to the solvers that use the simplex method.

In the following example, the top face of the cube is optimal, and CVXOPT gives the center point of the top face, whereas the other tested solvers return a vertex:
**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()`.

**EXAMPLES:**

```python
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)
True
```

**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.

**EXAMPLES:**

```python
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=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.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
sage: p.add_variable()
0
```
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.

The following backend is used for debugging and testing purposes.

## 11.8 Logging Backend

It records, for debugging and unit testing purposes, all calls to backend methods in one of three ways.

See [LoggingBackendFactory](#) for more information.

```python
class sage.numerical.backends.logging_backend.LoggingBackend(backend, printing=True, doctest=None, test_method=None, base_ring=None)
```

Bases: `GenericBackend`

See [LoggingBackendFactory](#) for documentation.

**EXAMPLES:**

```python
sage: import sage.numerical.backends.logging_backend
sage: from sage.numerical.backends.logging_backend import LoggingBackend
sage: from sage.numerical.backends.generic_backend import get_solver
sage: b = get_solver(solver = "GLPK")
sage: lb = LoggingBackend(backend=b)
sage: lb.add_variable(obj=42, name='Helloooooo')
# p.add_variable(obj=42, name='Helloooooo')
# result: 0
0
sage: lb.add_variable(obj=1789)
# p.add_variable(obj=1789)
# result: 1
1
```

**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.
EXAMPLES:

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.nrows()
0
sage: p.add_linear_constraints(5, 0, None)
```

```
sage: p.add_col(list(range(5)), list(range(5)))
sage: p.nrows()
5
```

**add_linear_constraint** \((\textit{coefficients}, \textit{lower_bound}, \textit{upper_bound}, \textit{name}=\text{None})\)

Add a linear constraint.

**INPUT:**

- \textit{coefficients} – an iterable of pairs \((i, v)\). In each pair, \(i\) is a variable index (integer) and \(v\) is a value (element of \textit{base_ring}()).
- \textit{lower_bound} – element of \textit{base_ring}() or None. The lower bound.
- \textit{upper_bound} – element of \textit{base_ring}() or None. The upper bound.
- \textit{name} – string or None. Optional name for this row.

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(5)
4
sage: p.add_linear_constraint( zip(range(5), range(5)), 2.0, 2.0)
sage: p.row(0)
([0, 1, 2, 3, 4], [0.0, 1.0, 2.0, 3.0, 4.0])
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_constraint_vector** \((\textit{degree}, \textit{coefficients}, \textit{lower_bound}, \textit{upper_bound}, \textit{name}=\text{None})\)

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:**

- \textit{degree} – integer. The vector degree, that is, the number of new scalar constraints.
- \textit{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 \textit{degree}).
- \textit{lower_bound} – either a vector or None. The component-wise lower bound.
- \textit{upper_bound} – either a vector or None. The component-wise upper bound.
• name – string or None. An optional name for all new rows.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
coeffs = ([0, vector([1, 2])], [1, vector([2, 3])])
```

```python
upper = vector([5, 5])
```

```python
lower = vector([0, 0])
```

```python
p.add_variables(2)
```

```python
p.add_linear_constraint_vector(2, coeffs, lower, upper, '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)

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
p.add_variables(5)
```

```python
p.add_linear_constraints(5, None, 2)
```

```python
p.row(4)
```

```python
([], [])
```

```python
p.row_bounds(4)
```

```python
(None, 2.0)
```

add_variable (*args, **kwargs)

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 continuous (default: True).
• integer - True if the variable is integral (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

EXAMPLES:
```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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** (*args, **kwargs)

Add \(n\) variables.

This amounts to adding new columns to the matrix. By default, the variables are both nonnegative 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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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
```
**base_ring()**

Return the base ring.

The backend’s base ring can be overridden. It is best to run the tests with GLPK and override the base ring to QQ. Then default input to backend methods, prepared by `MixedIntegerLinearProgram`, depends on the base ring. This way input will be rational and so suitable for both exact and inexact methods; whereas output will be float and will thus trigger `assertAlmostEqual()` tests.

**EXAMPLES:**

```python
sage: import sage.numerical.backends.logging_backend
sage: from sage.numerical.backends.logging_backend import LoggingBackend
sage: from sage.numerical.backends.generic_backend import get_solver
sage: b = get_solver(solver = "GLPK")
sage: lb = LoggingBackend(backend=b)
Real Double Field
sage: from sage.rings.rational_field import QQ
sage: lb = LoggingBackend(backend=b, base_ring=QQ)
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 `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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: b = p.new_variable(binary=True)
sage: for u,v in graphs.CycleGraph(5).edges(labels=False):
....:     p.add_constraint(b[u]+b[v]<=1)
sage: p.set_objective(p.sum(b[x] for x in range(5)))
sage: p.solve()
2.0
sage: pb = p.get_backend()
sage: pb.get_objective_value()
2.0
sage: pb.best_known_objective_bound()
2.0
```

**category()**

**col_bounds(index)**

Return the bounds of a specific variable.

**INPUT:**

- **index** (integer) – the variable’s id.

**OUTPUT:**
A pair \((\text{lower\_bound}, \text{upper\_bound})\). Each of them can be set to \text{None} if the variable is not bounded in the corresponding direction, and is a real value otherwise.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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 \((\text{index})\)

Return the index-th column name

INPUT:

- \text{index} (integer) – the column id
- \text{name} (char *) – its name. When set to NULL (default), the method returns the current name.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variable(name="I am a variable")
1	sage: p.col_name(0)
'I am a variable'
```

copy ()

Returns a copy of self.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: b = p.new_variable()
sage: p.set_objective(b[1] + b[2])
sage: copy(p).solve()
6.0
```

dump \((\text{filename}, \text{compress=\text{True}})\)

Same as self.save(filename, compress)

dumps \((\text{compress=\text{True}})\)

Dump self to a string \(s\), which can later be reconstituted as self using loads \((s)\).

There is an optional boolean argument \text{compress} which defaults to \text{True}.

EXAMPLES:
sage: from sage.misc.persist import comp
sage: O = SageObject()

sage: p_comp = O.dumps()
sage: p_uncomp = O.dumps(compress=False)
sage: comp.decompress(p_comp) == p_uncomp
True
sage: import pickletools
sage: pickletools.dis(p_uncomp)
  0: \x80 PROTO 2
  2: c GLOBAL 'sage.structure.sage_object SageObject'
  41: q BINPUT ...
  43: ) EMPTY_TUPLE
  44: \x81 NEWOBJ
  45: q BINPUT ...
  47: . STOP

highest protocol among opcodes = 2

g et_custom_name()

Return the custom name of this object, or None if it is not renamed.

EXAMPLES:

sage: P.<x> = QQ[]

sage: P.get_custom_name() is None
True
sage: P.rename('A polynomial ring')

sage: P.get_custom_name()
'A polynomial ring'

sage: P.reset_name()

sage: P.get_custom_name() is None
True

g et_objective_value()

Return the value of the objective function.

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

EXAMPLES:

sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver

sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variables(2)

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)  
0.0

sage: p.get_variable_value(1)  
1.5

g et_relative_objective_gap()
Return the relative objective gap of the best known solution.

For a minimization problem, this value is computed by \( \frac{\text{best integer} - \text{best objective}}{10 + |\text{best objective}|} \), where \text{best integer} is the value returned by \text{get_objective_value()} and \text{best objective} is the value returned by \text{best_known_objective_bound()}. For a maximization problem, the value is computed by \( \frac{\text{best objective} - \text{best integer}}{10 + |\text{best objective}|} \).

\textbf{Note:} Has no meaning unless \text{solve} has been called before.

\begin{verbatim}
EXAMPLES:

sage: # optional - nonexistent_LP_solver
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
Ideal: Variables
ideal: slack
ideal: obj
sage: b = p.new_variable(binary=True)
Ideal: Variables
ideal: slack
ideal: obj
ideal: b
sage: for u,v in graphs.CycleGraph(5).edges(labels=False):
    ....:     p.add_constraint(b[u]+b[v]<=1)
Ideal: Constraints
ideal: c0 : b[0] + b[1] <= 1
ideal: c4 : b[4] + b[0] <= 1
sage: p.set_objective(p.sum(b[x] for x in range(5)))
Ideal: Objective
ideal: obj
sage: p.solve()
2.0
sage: pb = p.get_backend()
Ideal: Variables
ideal: slack
ideal: obj
ideal: b
sage: pb.get_objective_value()
2.0
sage: pb.get_relative_objective_gap()
0.0
\end{verbatim}

\textbf{get_variable_value} (\textit{variable})

Return the value of a variable given by the solver.

\textbf{Note:} Behavior is undefined unless \text{solve} has been called before.

\begin{verbatim}
EXAMPLES:

sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
Ideal: Variables
ideal: slack
ideal: obj
sage: p.add_variables(2)
Ideal: Variables
ideal: slack
ideal: obj
ideal: x
ideal: y
sage: p.add_linear_constraint([(0,1), (1, 2)], None, 3)
Ideal: Constraints
ideal: c0 : 0*x + 1*y <= 3
sage: p.set_objective([2, 5])
Ideal: Objective
ideal: obj
sage: p.solve()
0
sage: p.get_objective_value()
7.5
sage: p.get_variable_value(0)
0.0
sage: p.get_variable_value(1)
1.5
\end{verbatim}

\textbf{is_maximization} (*\textit{args}, **\textit{kwdargs})

Test whether the problem is a maximization

\begin{verbatim}
EXAMPLES:
\end{verbatim}
is_slack_variable_basic(*args, **kwdargs)
Test whether the slack variable of the given row is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
          solver="Nonexistent LP solver")
sage: x = p.new_variable(nonnegative=True)
```

```
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
sage: b.solve() 0
sage: b.is_slack_variable_basic(0)
True
sage: b.is_slack_variable_basic(1)
False
```

is_slack_variable_nonbasic_at_lower_bound(*args, **kwdargs)
Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```
sage: # optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
          solver="Nonexistent LP solver")
```
is_variable_basic (*args, **kwds)
Test whether the given variable is basic.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

- index (integer) – the variable’s id

EXAMPLES:

```sage
# optional - nonexistent_lp_solver
sage: p = MixedIntegerLinearProgram(maximization=True,
....: solver="Nonexistent_LP_solver")
sage: x = p.new_variable(nonnegative=True)
sage: p.add_constraint(-x[0] + x[1] <= 2)
sage: p.add_constraint(8 * x[0] + 2 * x[1] <= 17)
sage: p.set_objective(5.5 * x[0] - 3 * x[1])
sage: b = p.get_backend()
```

```here
sage: b.solve()
0
```

```sage
sage: b.is_slack_variable_nonbasic_at_lower_bound(0)
False
sage: b.is_slack_variable_nonbasic_at_lower_bound(1)
True
```

is_variable_binary (*args, **kwds)
Test whether the given variable is of binary type.

INPUT:

- index (integer) – the variable’s id

EXAMPLES:

```sage
# optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```here
sage: b.solve()
0
```

```sage
sage: b.is_variable_binary(0)
True
sage: b.is_variable_binary(1)
False
```

is_variable_continuous (*args, **kwds)
Test whether the given variable is of continuous/real type.
INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_continuous(0)
True
sage: p.set_variable_type(0,1)
None
sage: p.is_variable_continuous(0)
False
```

`is_variable_integer(*args, **kwds)`

Test whether the given variable is of integer type.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.is_variable_continuous(0)
True
sage: p.set_variable_type(0,1)
None
sage: p.is_variable_continuous(0)
False
```

`is_variable_nonbasic_at_lower_bound(*args, **kwds)`

Test whether the given variable is nonbasic at lower bound.

This assumes that the problem has been solved with the simplex method and a basis is available. Otherwise an exception will be raised.

INPUT:

• index (integer) – the variable’s id

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variable()
0
sage: p.set_variable_type(0,1)
None
sage: p.is_variable_integer(0)
True
```

(continues on next page)
```python
--here
sage: b.solve()
0
sage: b.is_variable_nonbasic_at_lower_bound(0)
False
sage: b.is_variable_nonbasic_at_lower_bound(1)
True
```

**ncols (**args**, **kwargs**)**

Return the number of columns/variables.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variables(2)
1
sage: p.ncols()
2
```

**nrows (**args**, **kwargs**)**

Return the number of rows/constraints.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.nrows()
0
sage: p.add_linear_constraints(2, 2.0, None)
1
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

EXAMPLES:

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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
```
objective_constant_term \(d=\text{None}\)

Set or get the constant term in the objective function.

INPUT:

- \(d\) (double) – its coefficient. If \(\text{None}\) (default), return the current value.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
sage: p.objective_constant_term()
0.0
```

```python
sage: p.objective_constant_term(42)
```

```python
sage: p.objective_constant_term()
42.0
```

parent()

Return the type of \(\text{self}\) to support the coercion framework.

EXAMPLES:

```python
sage: t = \log(\sqrt{2} - 1) + \log(\sqrt{2} + 1); t
# needs sage.symbolic
```

```python
\log(\sqrt{2} + 1) + \log(\sqrt{2} - 1)
```

```python
sage: u = t.maxima_methods()  # needs sage.symbolic
sage: u.parent()  # needs sage.symbolic
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
```

problem_name \(\text{name=\text{None}}\)

Return or define the problem’s name.

INPUT:

- \text{name} (str) – the problem’s name. When set to \text{None} (default), the method returns the problem’s name.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")  # optional - Nonexistent_LP_solver
```

```python
sage: p.problem_name("There once was a french fry")  # optional - Nonexistent_LP_solver
```

```python
sage: print(p.problem_name())  # optional - Nonexistent_LP_solver
```

remove_constraint \(i\)

Remove a constraint.

INPUT:

- \(i\) – index of the constraint to remove.

EXAMPLES:
```python
sage: # optional - nonexistent_LP_solver
sage: p = MixedIntegerLinearProgram(solver="Nonexistent_LP_solver")
sage: v = p.new_variable(nonnegative=True)
sage: x, y = v[0], v[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
sage: p.remove_constraint(0)
sage: p.solve()
10.0
sage: p.get_values([x,y])
[0.0, 3.0]
```

`remove_constraints(constraints)`

Remove several constraints.

**INPUT:**

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

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(2)
1
sage: p.add_linear_constraint([(0, 2), (1, 3)], None, 6)
sage: p.add_linear_constraint([(0, 3), (1, 2)], None, 6)
sage: p.remove_constraints([0, 1])
```

`rename(x=None)`

Change self so it prints as x, where x is a string.

If x is None, the existing custom name is removed.

**Note:** This is *only* supported for Python classes that derive from `SageObject`.

**EXAMPLES:**

```python
sage: x = PolynomialRing(QQ, 'x', sparse=True).gen()
sage: g = x^3 + x - 5
sage: g
x^3 + x - 5
sage: g.rename('a polynomial')
sage: g
a polynomial
sage: g + x
x^3 + 2*x - 5
sage: h = g^100
sage: str(h)[:20]
'x^300 + 100*x^298 - '
```

Real numbers are not Python classes, so rename is not supported:

```
sage: a = 3.14
sage: type(a)  #→
˓→needs sage.rings.real_mpfr
<... sage.rings.real_mpfr.RealLiteral>
sage: a.rename('pi')  #→
˓→needs sage.rings.real_mpfr
Traceback (most recent call last):
  ...
NotImplementedError: object does not support renaming: 3.14000000000000
```

**Note:** The reason C-extension types are not supported by default is if they were then every single one would have to carry around an extra attribute, which would be slower and waste a lot of memory.

To support them for a specific class, add a `cdef public _SageObject__custom_name` attribute.

### reset_name()
Remove the custom name of an object.

**EXAMPLES:**

```
sage: P.<x> = QQ[]
sage: P
Univariate Polynomial Ring in x over Rational Field
sage: P.rename('A polynomial ring')
sage: P
A polynomial ring
sage: P.reset_name()
sage: P
Univariate Polynomial Ring in x over Rational Field
```

### 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.

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(5)
```

(continues on next page)
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.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_linear_constraints(1, 2, None, names=['Empty constraint 1'])
sage: p.row_name(0)
'Empty constraint 1'
```

row_name(index)

Return the index th row name

INPUT:

• index (integer) – the row’s id

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_linear_constraints(1, 2, None, names=['Empty constraint 1'])
sage: p.row_name(0)
'Empty constraint 1'
```

save(filename=None, compress=True)

Save self to the given filename.

EXAMPLES:

```python
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".sobj") as t:
    ....:     f.save(t.name)
```

(continues on next page)
 Numerical Optimization, Release 10.3

....: load(t.name)
x^3 + 5

**set_objective**(coef, d=0.0)
Set the objective function.

**INPUT:**

- coef – 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)

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
sage: p.add_variables(5)
4
sage: p.set_objective([1, 1, 2, 1, 3])
```

```python
sage: [p.objective_coefficient(x) for x in range(5)]
[1.0, 1.0, 2.0, 1.0, 3.0]
```

Constants in the objective function are respected:

```python
sage: # optional - nonexistent_LP_solver
sage: p = MixedIntegerLinearProgram(solver=Nonexistent_LP_solver)
```

```python
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()
```

**set_sense**(sense)
Set the direction (maximization/minimization).

**INPUT:**

- sense (integer):
  - +1 => Maximization
  - -1 => Minimization

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```python
sage: p.is_maximization()
```

```python
sage: p.set_sense(-1)
```

```python
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 Continuous

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```
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 log (verbosity) level

**INPUT:**

- *level* (integer) – From 0 (no verbosity) to 3.

**EXAMPLES:**

```
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```
sage: p.set_verbosity(2)
```

**solve** (*args*, **kwdargs**)

Solve the problem.

**Note:** This method raises MIPSolverException exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc…)

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_linear_constraints(5, 0, None)
sage: p.add_col(list(range(5)), list(range(5)))
```
```
sage: p.solve()
0
```
```
sage: p.objective_coefficient(0,1)
```
(continues on next page)
sage: p.solve()
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()`.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.solver_parameter("timelimit")

sage: p.solver_parameter("timelimit", 60)

sage: p.solver_parameter("timelimit")
```

**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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

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)
```

**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.

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

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)
(5.0, None)
```
```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```
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)
```

**write_lp**(name)

Write the problem to a .lp file

**INPUT:**

- filename (string)

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```

```
sage: p.add_variables(2) 2
sage: p.add_linear_constraint([(0, 1], (1, 2)], None, 3)
sage: p.set_objective([2, 5])
```

```
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".lp") as f:
....:    p.write_lp(f.name)
```

**write_mps**(name, modern)

Write the problem to a .mps file

**INPUT:**

- filename (string)

**EXAMPLES:**

```
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_backend import get_solver
```

```
sage: p.add_variables(2) 2
sage: p.add_linear_constraint([(0, 1], (1, 2)], None, 3)
sage: p.set_objective([2, 5])
```

```
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".lp") as f:
....:    p.write_lp(f.name)
```

```
sage.numerical.backends.logging_backend.LoggingBackendFactory(solver=None,
    printing=True,
    doctest_file=None,
    test_method_file=None,
    test_method=None,
    base_ring=Ring)
```

Factory that constructs a `LoggingBackend` for debugging and testing.
An instance of it can be passed as the solver argument of `sage.numerical.backends.generic_backend.get_solver()` and `MixedIntegerLinearProgram`.

EXAMPLES:

Assume that we have the following function that does some computation using `MixedIntegerLinearProgram` (or MIP backend methods), and suppose we have observed that it works with the GLPK backend, but not with the COIN backend:

```python
sage: def compute_something(solver=GLPK):
    ....:     from sage.numerical.mip import MIPSolverException
    ....:     mip = MixedIntegerLinearProgram(solver=solver)
    ....:     lb = mip.get_backend()
    ....:     lb.add_variable(obj=42, name='Helloooooo')
    ....:     lb.add_variable(obj=1789)
    ....:     try:
    ....:         lb.solve()
    ....:     except MIPSolverException:
    ....:         return 4711
    ....:     else:
    ....:         return 91
```

We can investigate what the backend methods are doing by running a `LoggingBackend` in its in-terminal logging mode:

```python
sage: import sage.numerical.backends.logging_backend
sage: from sage.numerical.backends.logging_backend import LoggingBackendFactory
sage: compute_something(solver = LoggingBackendFactory(solver=GLPK))
# p = get_solver(solver=GLPK)
# p.add_variable(obj=42, name='Helloooooo')
# result: 0
# p.add_variable(obj=1789)
# result: 1
# p.solve()
# exception: GLPK: The LP (relaxation) problem has no dual feasible solution
4711
```

By replacing ‘GLPK’ by ‘COIN’ above, we can then compare the two logs and see where they differ.

Imagine that we have now fixed the bug in the COIN backend, and we want to add a doctest that documents this fact. We do not want to call `compute_something` in the doctest, but rather just have a sequence of calls to backend methods.

We can have the doctest autogenerated by running a `LoggingBackend` in its doctest-writing mode:

```python
sage: fname = tmp_filename()
sage: compute_something(solver = LoggingBackendFactory(solver='GLPK',
    ....:     printing=False,
    ....:     doctest_file=fname))
4711
sage: with open(fname) as f:
    ....:     for line in f.readlines(): _ = sys.stdout.write('|{:.}.format(line))
| sage: p = get_solver(solver='GLPK')
| sage: p.add_variable(obj=42, name='Helloooooo')
| 0
| sage: p.add_variable(obj=1789)
| 1
| sage: p.solve()
| Traceback (most recent call last):
```

(continues on next page)
We then copy from the generated file and paste into the source code of the COIN backend.

If this test seems valuable enough that all backends should be tested against it, we should create a test method instead of a docstring.

We can have the test method autogenerated by running a `LoggingBackend` in its test-method-writing mode:

```sage
def _test_something(cls, tester=None, **options):
    ...:
    Run tests on ...

    TESTS:
    ...

    @classmethod
    def _test_something(cls, tester=None, **options):
        ...
        Run tests on ...

        TESTS:
        ...

        @classmethod
        def _test_something(cls, tester=None, **options):
            ...
            Run tests on ...

            TESTS:
            ...
```

```
sage: with open(fname) as f:
    ....: for line in f.readlines(): _ = sys.stdout.write('{}'.format(line))
```

```
sage: from sage.numerical.backends.generic_backend import GenericBackend
sage: p = GenericBackend()
sage: p._test_something()
Traceback (most recent call last):
  ...
NotImplementedError
```

We then copy from the generated file and paste into the source code of the generic backend, where all test methods are defined.

If `test_method_file` is not provided, a default output file name will be computed from `test_method`.
SEMIDEFINITE OPTIMIZATION (SDP) SOLVER BACKENDS

12.1 Generic Backend for SDP solvers

This class only lists the methods that should be defined by any interface with a SDP 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_SDP_solver" by the solver's name, and replace `GenericSDPBackend` by `SolverName(GenericSDPBackend)` so that the new solver extends this class.

AUTHORS:

- Ingolfur Edvardsson (2014-07): initial implementation

```python
class sage.numerical.backends.generic_sdp_backend.GenericSDPBackend:
    Bases: object

    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).
        - 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)

    EXAMPLES:

    sage: # optional - nonexistent_lp_solver
    sage: from sage.numerical.backends.generic_sdp_backend import get_solver
    sage: p = get_solver(solver="Nonexistent_LP_solver")
    sage: p.add_variables(5)
    4
    sage: p.add_linear_constraint(zip(range(5), range(5)), 2.0, 2.0)
    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)
    sage: p.add_linear_constraint(zip(range(5), range(5)), 1.0, 1.0, name='foo')
    sage: p.row_name(-1)  # optional - Nonexistent_LP_solver
    "foo"
```


**add_linear_constraints** (*number, 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)

**EXAMPLES:**

```sage
# optional - nonexistent_lp_solver
from sage.numerical.backends.generic_sdp_backend import get_solver
p = get_solver(solver="Nonexistent_LP_solver")
p.add_variables(5)
5
p.add_linear_constraints(5, None, 2)
p.row(4)
([], [])
p.row_bounds(4)
(None, 2.0)
```

**add_variable** (*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 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

**EXAMPLES:**

```sage
# optional - nonexistent_lp_solver
from sage.numerical.backends.generic_sdp_backend import get_solver
p = get_solver(solver="Nonexistent_LP_solver")
p.ncols()
0
p.add_variable()
0
p.ncols()
1
p.add_variable(name='x', obj=1.0)
3
p.col_name(3)
'x'
p.objective_coefficient(3)
1.0
```

**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)
• \( \text{obj} \) - (optional) coefficient of all variables in the objective function (default: 0.0)
• \( \text{names} \) - optional list of names (default: None)

OUTPUT: The index of the variable created last.

EXAMPLES:

```python
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")

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
```

**base_ring**

The base ring

**col_name(index)**

Return the \( \text{index} \) th col name

INPUT:

• \( \text{index} \) (integer) – the col’s id
• \( \text{name} \) (char *) – its name. When set to NULL (default), the method returns the current name.

EXAMPLES:

```python
sage: p = get_solver(solver="Nonexistent_LP_solver")

sage: p.add_variable(name="I am a variable")
1
sage: p.col_name(0)
'I am a variable'
```

**dual_variable(i, sparse=False)**

The \( i \)-th dual variable

Available after self.solve() is called, otherwise the result is undefined

• \( \text{index} \) (integer) – the constraint’s id.

OUTPUT:

The matrix of the \( i \)-th dual variable

EXAMPLES:

```python
sage: p = SemidefiniteProgram(maximization=False, solver="Nonexistent_LP_solver")

sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1])
```

(continues on next page)
```python
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.solve()
-3.0
sage: B = p.get_backend()
sage: x = p.get_values(x).values()
sage: -(a3*B.dual_variable(0)).trace()-(b3*B.dual_variable(1)).trace()
-3.0
sage: g = sum((B.slack(j)*B.dual_variable(j)).trace() for j in range(2)); g
0.0
```

**get_objective_value()**

Return the value of the objective function.

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

**EXAMPLES:**

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variables(2)
2
sage: p.add_linear_constraint(((0,1), (1,2)), None, 3)

```

**get_variable_value(variable)**

Return the value of a variable given by the solver.

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

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
2
sage: p.add_linear_constraint(((0,1), (1, 2)), None, 3)
```

(continues on next page)
is_maximization()

Test whether the problem is a maximization

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.is_maximization()
True
sage: p.set_sense(-1)
```

ncols()

Return the number of columns/variables.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.ncols()
0
sage: p.add_variables(2)
2
sage: p.ncols()
2
```

nrows()

Return the number of rows/constraints.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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 (double) – its coefficient

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.add_variable(1)
```
```python
sage: p.objective_coefficient(0)
0.0
sage: p.objective_coefficient(0, 2)
```
```python
sage: p.objective_coefficient(0)
2.0
```

problem_name (name=None)
Return or define the problem’s name

INPUT:
• name (str) – the problem’s name. When set to NULL (default), the method returns the problem’s name.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```python
sage: p.problem_name("There once was a french fry")
```
```python
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.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```python
sage: p.add_variables(5)
5
```
```python
sage: p.add_linear_constraint(zip(range(5), range(5)), 2, 2)
```
```python
sage: p.row(0)
([4, 3, 2, 1], [4.0, 3.0, 2.0, 1.0])
```
```python
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

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```
sage: p.add_linear_constraints(1, 2, None, name="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)

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```
sage: p.add_variables(5)
```
```
sage: p.set_objective([1, 1, 2, 1, 3])
```
```
[p.objective_coefficient(x) for x in range(5)]
```
```
[1.0, 1.0, 2.0, 1.0, 3.0]
```

Constants in the objective function are respected.

**set_sense** (sense)

Set the direction (maximization/minimization).

INPUT:

• sense (integer):
  - +1 => Maximization
  - -1 => Minimization

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
```
```
sage: p.is_maximization()
True
```
```
sage: p.set_sense(-1)
```
```
sage: p.is_maximization()
False
```

**slack** (i, sparse=False)

Slack of the i-th constraint

Available after self.solve() is called, otherwise the result is undefined
• **index** (integer) – the constraint’s id.

**OUTPUT:**

The matrix of the slack of the $i$-th constraint

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: p = SemidefiniteProgram(maximization=False, solver="Nonexistent_LP_solver")
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.solve()
-3.0
sage: B = p.get_backend()
sage: B1 = B.slack(1); B1
[0.0 0.0]
[0.0 0.0]
sage: B1.is_positive_definite()
True
sage: x = p.get_values(x).values()
sage: x[0]*b1 + x[1]*b2 - b3 + B1
[0.0 0.0]
[0.0 0.0]
```

**solve()**

Solve the problem.

**Note:** This method raises `SDPSolverException` exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

**EXAMPLES:**

```python
sage: # optional - nonexistent_LP_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
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):
  ... SDPSolverException: ...
```

**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()`.

EXAMPLES:

```python
sage: # optional - nonexistent_lp_solver
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="Nonexistent_LP_solver")
sage: p.solver_parameter("timelimit")
sage: p.solver_parameter("timelimit", 60)
sage: p.solver_parameter("timelimit")
```

**zero()**

Zero of the base ring

`sage.numerical.backends.generic_sdp_backend.default_sdp_solver(solver=None)`

Return/set the default SDP solver used by Sage

INPUT:

- **solver** – one of the following:
  - the string "CVXOPT", to make the use of the CVXOPT solver (see the CVXOPT web site) the default;
  - a subclass of `sage.numerical.backends.generic_sdp_backend.GenericSDPBackend`, to make it the default; or
  - None (default), in which case the current default solver (a string or a class) is returned.

OUTPUT:

This function returns the current default solver (a string or a class) 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.

EXAMPLES:

```python
sage: former_solver = default_sdp_solver()
```

```python
sage: default_sdp_solver("Cvxopt")
```

```python
sage: default_sdp_solver() 'Cvxopt'
```

```python
sage: default_sdp_solver("Yeahhhhhhhhhhh")
```

Traceback (most recent call last):
...
`ValueError: 'solver' should be set to ...
```

```python
sage: default_sdp_solver(former_solver)
```

```python
sage: from sage.numerical.backends.generic_sdp_backend import GenericSDPBackend
sage: class my_sdp_solver(GenericSDPBackend): pass
```

```python
sage: default_sdp_solver(my_sdp_solver)
```

```python
sage: default_sdp_solver() is my_sdp_solver
```

**True**

`sage.numerical.backends.generic_sdp_backend.get_solver(solver=None, base_ring=None)`

Return a solver according to the given preferences.

INPUT:
• **solver** – one of the following:
  – the string "CVXOPT", designating the use of the CVXOPT solver (see the CVXOPT website);
  – a subclass of `sage.numerical.backends.generic_sdp_backend.GenericSDPBackend`;
  – None (default), in which case the default solver is used (see `default_sdp_solver()`);

See also:

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

EXAMPLES:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver()
```

Passing a class:

```
sage: from sage.numerical.backends.generic_sdp_backend import GenericSDPBackend
sage: class MockSDPBackend(GenericSDPBackend):
    ....:     def solve(self):
    ....:         raise RuntimeError("SDP is too slow")

sage: P = SemidefiniteProgram(solver=MockSDPBackend)
sage: P.solve()
Traceback (most recent call last):
  ...  RuntimeError: SDP is too slow
```

### 12.2 CVXOPT SDP Backend

AUTHORS:

• Ingolfur Edvardsson (2014-05): initial implementation
• Dima Pasechnik (2015-12): minor fixes

```python
class sage.numerical.backends.cvxopt_sdp_backend.CVXOPTSDPBackend
    Bases: MatrixSDPBackend
    Cython constructor
```

EXAMPLES:

```
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
```

`dual_variable(i, sparse=False)`

The `i`-th dual variable

Available after `self.solve()` is called, otherwise the result is undefined

• `index` (integer) – the constraint’s id.

OUTPUT:

The matrix of the `i`-th dual variable

EXAMPLES:
sage: p = SemidefiniteProgram(maximization=False, solver='cvxopt')
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.solve()    # tol 1e-08
-3.0
sage: B = p.get_backend()
sage: x = p.get_values(x).values()
sage: -(a3*B.dual_variable(0)).trace() - (b3*B.dual_variable(1)).trace()
    # tol 1e-07
-3.0
sage: g = sum((B.slack(j)*B.dual_variable(j)).trace() for j in range(2)); g
    # tol 1.5e-08
0.0

get_objective_value()

Return the value of the objective function.

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

EXAMPLES:

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: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
sage: N(p.solve(), digits=4)
-3.154
sage: N(p.get_backend().get_objective_value(), digits=4)
-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.

EXAMPLES:
```
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: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
sage: N(p.solve(), digits=4)
-3.154
sage: N(p.get_backend().get_variable_value(0), digits=3)
-0.368
sage: N(p.get_backend().get_variable_value(1), digits=4)
1.898
sage: N(p.get_backend().get_variable_value(2), digits=3)
-0.888
```

**slack** *(i, sparse=False)*

Slack of the *i*-th constraint

Available after `self.solve()` is called, otherwise the result is undefined

- **index** (integer) – the constraint’s id.

**OUTPUT:**

The matrix of the slack of the *i*-th constraint

**EXAMPLES:**

```
sage: p = SemidefiniteProgram(maximization = False, solver='cvxopt')
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.solve()  # tol 1e-08
-3.0
sage: B = p.get_backend()
sage: B1 = B.slack(1); B1  # tol 1e-08
[0.0 0.0]
[0.0 0.0]
sage: B1.is_positive_definite()
True
sage: x = sorted(p.get_values(x).values())
sage: x[0]*b1 + x[1]*b2 - b3 + B1  # tol 1e-09
[0.0 0.0]
[0.0 0.0]
```
solve()

Solve the problem.

**Note:** This method raises :class:`SDPSolverException` exceptions when the solution cannot be computed for any reason (none exists, or the LP solver was not able to find it, etc...)

**EXAMPLES:**

```python
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.], [-6., 8.], [-17., 8., 6.]])
sage: p.add_constraint(a1*x[0] + a3*x[2] <= a4)
sage: p.add_constraint(b1*x[0] + b3*x[2] <= b4)
sage: N(p.solve(), digits=4)
-3.225
```

```python
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.], [-6., 8.], [-17., 8., 6.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
sage: N(p.solve(), digits=4)
-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 :meth:`solver_parameter()`.

**EXAMPLES:**

```python
sage: from sage.numerical.backends.generic_sdp_backend import get_solver
sage: p = get_solver(solver="CVXOPT")
sage: p.solver_parameter("show_progress")
```

(continues on next page)
For more details on CVXOPT, see CVXOPT documentation.
CHAPTER
THIRTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page
## PYTHON MODULE INDEX

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.numerical.backends.cvxopt_backend</td>
<td>232</td>
</tr>
<tr>
<td>sage.numerical.backends.cvxopt_sdp_backend</td>
<td>274</td>
</tr>
<tr>
<td>sage.numerical.backends.generic_backend</td>
<td>143</td>
</tr>
<tr>
<td>sage.numerical.backends.generic_sdp_backend</td>
<td>265</td>
</tr>
<tr>
<td>sage.numerical.backends.glpk_backend</td>
<td>175</td>
</tr>
<tr>
<td>sage.numerical.backends.glpk_exact_backend</td>
<td>205</td>
</tr>
<tr>
<td>sage.numerical.backends.glpk_graph_backend</td>
<td>208</td>
</tr>
<tr>
<td>sage.numerical.backends.interactivelp_backend</td>
<td>162</td>
</tr>
<tr>
<td>sage.numerical.backends.logging_backend</td>
<td>243</td>
</tr>
<tr>
<td>sage.numerical.backends.ppl_backend</td>
<td>220</td>
</tr>
<tr>
<td>sage.numerical.gauss_legendre</td>
<td>139</td>
</tr>
<tr>
<td>sage.numerical.interactive_simplex_method</td>
<td>83</td>
</tr>
<tr>
<td>sage.numerical.knapsack</td>
<td>1</td>
</tr>
<tr>
<td>sage.numerical.linear_functions</td>
<td>53</td>
</tr>
<tr>
<td>sage.numerical.linear_tensor</td>
<td>63</td>
</tr>
<tr>
<td>sage.numerical.linear_tensor_constraints</td>
<td>69</td>
</tr>
<tr>
<td>sage.numerical.linear_tensor_element</td>
<td>67</td>
</tr>
<tr>
<td>sage.numerical.mip</td>
<td>7</td>
</tr>
<tr>
<td>sage.numerical.optimize</td>
<td>73</td>
</tr>
<tr>
<td>sage.numerical.sdp</td>
<td>39</td>
</tr>
</tbody>
</table>
A

A() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 86
A() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 127
A_N() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 127
Abcx() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 87
add_col() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 232
add_col() (sage.numerical.backends.generic_backend.GenericBackend method), 143
add_col() (sage.numerical.backends.glpk_backend.GLPKBackend method), 175
add_col() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 162
add_col() (sage.numerical.backends.logging_backend.LoggingBackend method), 243
add_col() (sage.numerical.backends.ppl_backend.PPLBackend method), 221
add_linear_constraint() (sage.numerical.backends.generic_backend.GenericBackend method), 144
add_linear_constraint() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 265
add_linear_constraint() (sage.numerical.backends.glpk_backend.GLPKBackend method), 175
add_linear_constraint() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 163
add_linear_constraint() (sage.numerical.backends.logging_backend.LoggingBackend method), 244
add_linear_constraint() (sage.numerical.backends.ppl_backend.PPLBackend method), 221
add_linear_constraint_vector() (sage.numerical.backends.generic_backend.GenericBackend method), 144
add_linear_constraint_vector() (sage.numerical.backends.logging_backend.LoggingBackend method), 244
add_linear_constraints() (sage.numerical.backends.generic_backend.GenericBackend method), 145
add_linear_constraints() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 265
add_linear_constraints() (sage.numerical.backends.glpk_backend.GLPKBackend method), 176
add_linear_constraints() (sage.numerical.backends.logging_backend.LoggingBackend method), 245
add_linear_constraints() (sage.numerical.backends.ppl_backend.PPLBackend method), 222
add_row() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 108
add_row() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 122
add_row() (sage.numerical.interactive_simplex_method.LPDictionary method), 108
add_row() (sage.numerical.interactive_simplex_method.LPDictionary method), 122
add_row() (sage.numerical.interactive_simplex_method.LPDictionary method), 108
plex_method.LPRevisedDictionary method), 129
add_variable() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 233
add_variable() (sage.numerical.backends.generic_backend.GenericBackend method), 145
add_variable() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 266
add_variable() (sage.numerical.backends.glpk_backend.GLPKBackend method), 176
add_variable() (sage.numerical.backends.glpk_exact_backend.GLPKExactBackend method), 205
add_variable() (sage.numerical.backends.interactive_backend.InteractiveLPBackend method), 163
add_variable() (sage.numerical.backends.logging_backend.LoggingBackend method), 245
add_variable() (sage.numerical.backends.ppl_backend.PPLBackend method), 222
add_variables() (sage.numerical.backends.generic_backend.GenericBackend method), 146
add_variables() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 266
add_variables() (sage.numerical.backends.glpk_backend.GLPKBackend method), 177
add_variables() (sage.numerical.backends.glpk_exact_backend.GLPKExactBackend method), 206
add_variables() (sage.numerical.backends.logging_backend.LoggingBackend method), 246
add_variables() (sage.numerical.backends.ppl_backend.PPLBackend method), 223
add_vertex() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 211
add_vertices() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 211
auxiliary_problem() (sage.numerical.interactive_simplex_method.InteractiveLPProblemStandardForm method), 101
auxiliary_variable() (sage.numerical.interactive_simplex_method.InteractiveLPProblemStandardForm method), 101
B() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 128
B_inverse() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 128
base_ring() (sage.numerical.backends.generic_backend.GenericBackend method), 147
base_ring() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 267
base_ring() (sage.numerical.backends.interactive_backend.InteractiveLPBackend method), 164
base_ring() (sage.numerical.backends.logging_backend.LoggingBackend method), 246
base_ring() (sage.numerical.backends.ppl_backend.PPLBackend method), 223
base_ring() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 88
base_ring() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 108
base_ring() (sage.numerical.mip.MixedIntegerLinearProgram method), 16
base_ring() (sage.numerical.sdp.SemidefiniteProgram method), 45
basic_indices() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 130
basic_solution() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 109
basic_variables() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 109
basic_variables() (sage.numerical.interactive_simplex_method.LPDictionary method), 122
basic_variables() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 130
best_known_objective_bound() (sage.numerical.backends.generic_backend.GenericBackend method), 147
best_known_objective_bound() (sage.numerical.backends.glpk_backend.GLPKBackend method), 178
best_known_objective_bound() (sage.numerical.backends.logging_backend.LoggingBackend method), 247
best_known_objective_bound() (sage.numerical.mip.MixedIntegerLinearProgram method), 17
binpacking() (in module sage.numerical.optimize), 73
constant_terms() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 89

D
decision_variables() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 90
default_mip_solver() (in module sage.numerical.backends.generic_backend), 160
default_sdp_solver() (in module sage.numerical.backends.generic_sdp_backend), 273
default_variable() (sage.numerical.mip.MixedIntegerLinearProgram method), 18
default_variable_name() (in module sage.numerical.interactive_simplex_method), 136
delete_edge() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 212
delete_edges() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 213
delete_vertex() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 212
get_backend()  (sage.numerical.sdp.SemidefiniteProgram method), 46
get_col_dual()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 181
get_col_stat()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 182
get_custom_name()  (sage.numerical.backends.logging_backend.LoggingBackend method), 249
get_edge()  (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 214
get_max()  (sage.numerical.mip.MixedIntegerLinearProgram method), 18
get_min()  (sage.numerical.mip.MixedIntegerLinearProgram method), 19
get_objective_value()  (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 235
get_objective_value()  (sage.numerical.backends.cvxopt_sdp_backend.CVXOPTSDPBackend method), 275
get_objective_value()  (sage.numerical.backends.generic_backend.GenericBackend method), 148
get_objective_value()  (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 268
get_objective_value()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 182
get_objective_value()  (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 165
get_objective_value()  (sage.numerical.backends.logging_backend.LoggingBackend method), 249
get_objective_value()  (sage.numerical.backends.ppl_backend.PPLBackend method), 224
get_objective_value()  (sage.numerical.mip.MixedIntegerLinearProgram method), 19
get_relative_objective_gap()  (sage.numerical.backends.generic_backend.GenericBackend method), 148
get_relative_objective_gap()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 183
get_relative_objective_gap()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 184
get_relative_objective_gap()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 184
get_relative_objective_gap()  (sage.numerical.backends.logging_backend.LoggingBackend method), 249
get_relative_objective_gap()  (sage.numerical.mip.MixedIntegerLinearProgram method), 19
get_row_dual()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 183
get_row_prim()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 184
get_row_stat()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 184
get_solver()  (in module sage.numerical.backends.generic_backend), 160
get_solver()  (in module sage.numerical.backends.generic_sdp_backend), 273
get_values()  (sage.numerical.mip.MixedIntegerLinearProgram method), 20
get_values()  (sage.numerical.sdp.SemidefiniteProgram method), 47
get_variable_value()  (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 235
get_variable_value()  (sage.numerical.backends.cvxopt_sdp_backend.CVXOPTSDPBackend method), 275
get_variable_value()  (sage.numerical.backends.generic_backend.GenericBackend method), 149
get_variable_value()  (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 268
get_variable_value()  (sage.numerical.backends.glpk_backend.GLPKBackend method), 185
get_variable_value()  (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 166
get_variable_value()  (sage.numerical.backends.logging_backend.LoggingBackend method), 250
get_variable_value()  (sage.numerical.backends.ppl_backend.PPLBackend method), 225
get_vertex()  (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 215
get_vertices()  (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 215
GLPKBackend  (class in sage.numerical.backends.glpk_backend), 175
GLPKExactBackend  (class in sage.numerical.backends.glpk_exact_backend), 205
GLPKGraphBackend  (class in sage.numerical.backends.glpk_graph_backend), 208
inequalities() (sage.numerical.linear_functions.LinearConstraint method), 55
init_mip() (sage.numerical.backends.ppl_backend.PPLBackend method), 225
initial_dictionary() (sage.numerical.interactive_simplex_method.InteractiveLPProblemStandardForm method), 104
inject_variables() (sage.numerical.interactive_simplex_method.InteractiveLPProblemStandardForm method), 104
integrate_vector() (in module sage.numerical.gauss_legendre), 140
integrate_vector_N() (in module sage.numerical.gauss_legendre), 140
interactive_lp_problem() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 166
interactive_lp_problem() (sage.numerical.mip.MixedIntegerLinearProgram method), 22
InteractiveLPBackend (class in sage.numerical.backends.interactivelp_backend), 162
InteractiveLPProblem (class in sage.numerical.interactive_simplex_method), 85
InteractiveLPProblemStandardForm (class in sage.numerical.interactive_simplex_method), 99
is_binary() (sage.numerical.mip.MixedIntegerLinearProgram method), 23
is_bounded() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 91
is_dual_feasible() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 112
is_equation() (sage.numerical.linear_functions.LinearConstraint method), 55
is_equation() (sage.numerical.linear_tensor_constraints.LinearTensorConstraint method), 70
is_feasible() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 91
is_feasible() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 113
is_integer() (sage.numerical.mip.MixedIntegerLinearProgram method), 23
is_less_or_equal() (sage.numerical.linear_functions.LinearConstraint method), 56
is_less_or_equal() (sage.numerical.linear_tensor_constraints.LinearTensorConstraint method), 70
isLinearConstraint() (in module sage.numerical.linear_functions), 61
is_LinearFunction() (in module sage.numerical.linear_functions), 61
is_LinearTensor() (in module sage.numerical.linear_tensor), 66
is_LinearTensorConstraint() (in module sage.numerical.linear_tensor_constraints), 72
is_matrix_space() (sage.numerical.linear_tensor.LinearTensorParent_class method), 65
is_maximization() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 235
is_maximization() (sage.numerical.backends.generic_backend.GenericBackend method), 269
is_maximization() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 269
is_maximization() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 166
is_maximization() (sage.numerical.backends.logging_backend.LoggingBackend method), 250
is_maximization() (sage.numerical.backends.ppl_backend.PPLBackend method), 225
is_negative() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 92
is_optimal() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 92
is_optimal() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 113
is_primal() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 92
is_real() (sage.numerical.mip.MixedIntegerLinearProgram method), 24
is_slack_variable_basic() (sage.numerical.backends.generic_backend.GenericBackend method), 150
is_slack_variable_basic() (sage.numerical.backends.glk_backend.GLPKBackend method), 186
is_slack_variable_basic() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 166
is_slack_variable_basic() (sage.numerical.backends.logging_backend.LoggingBackend method), 251
is_slack_variable_nonbasic_at_lower_bound() (sage.numerical...
Index 291

N

n() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 93
n_constraints() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 93
n_variables() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 94
ncols() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 237
ncols() (sage.numerical.backends.generic_backend.GenericBackend method), 153
ncols() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 269
ncols() (sage.numerical.backends.glpk_backend.GLPKBackend method), 189
ncols() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 254
ncols() (sage.numerical.backends.ppl_backend.PPLBackend method), 226
new_variable() (sage.numerical.mip.MixedIntegerLinearProgram method), 24
new_variable() (sage.numerical.sdp.SemidefiniteProgram method), 48
nodes() (in module sage.numerical.gauss_legendre), 141
nodes_uncached() (in module sage.numerical.gauss_legendre), 142
nonbasic_indices() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 132
nonbasic_variables() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 115
nonbasic_variables() (sage.numerical.interactive_simplex_method.LPDictionary method), 123
nonbasic_variables() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 133
nrows() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 237
nrows() (sage.numerical.backends.generic_backend.GenericBackend method), 153
nrows() (sage.numerical.backends.generic_sdp_backend_GenericSDPBackend method), 269
nrows() (sage.numerical.backends.glpk_backend.GLPKBackend method), 189
nrows() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 169
nrows() (sage.numerical.backends.logging_backend.LoggingBackend method), 254
nrows() (sage.numerical.backends.ppl_backend.PPLBackend method), 227
number_of_constraints() (sage.numerical.mip.MixedIntegerLinearProgram method), 26
number_of_constraints() (sage.numerical.sdp.SemidefiniteProgram method), 48
number_of_variables() (sage.numerical.mip.MixedIntegerLinearProgram method), 26
number_of_variables() (sage.numerical.sdp.SemidefiniteProgram method), 49

O

objective_coefficient() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 237
objective_coefficient() (sage.numerical.backends.generic_backend.GenericBackend method), 153
objective_coefficient() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 269
objective_coefficient() (sage.numerical.backends.glpk_backend.GLPKBackend method), 189
objective_coefficient() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 170
objective_coefficient() (sage.numerical.backends.logging_backend.LoggingBackend method), 254
objective_coefficient() (sage.numerical.backends.ppl_backend.PPLBackend method), 227
objective_coefficients() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 94
objective_coefficients() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 115
objective_coefficients() (sage.numerical.interactive_simplex_method.LPDictionary method), 123
objective_coefficients() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 133
objective_constant_term() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 94
objective_constant_term() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 115
objective_constant_term() (sage.numerical.interactive_simplex_method.LPDictionary method), 124
objective_constant_term() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 133
objective_name() (sage.numerical.interactive_simplex_method.InteractiveLPProblemStandardForm method), 105
objective_name() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 115
objective_name() (sage.numerical.interactive_simplex_method.LPDictionary method), 124
objective_name() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 133
objective_value() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 95
objective_value() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 116
objective_value() (sage.numerical.interactive_simplex_method.LPDictionary method), 124
objective_value() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 133
optimal_solution() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 95
optimal_value() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 95
plot() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 95
plot_feasible_set() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 96
polyhedron() (sage.numerical.mip.MixedIntegerLinearProgram method), 27
possible_dual_simplex_method_steps() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 116
possible_entering() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 116
possible_leaving() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 117
possible_simplex_method_steps() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 117
PPLBackend (class in sage.numerical.backends.ppl_backend), 227
print_ranges() (sage.numerical.backends.glpk_backend.GLPKBackend method), 190
problem() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 190
problem() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 134
problem_name() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 238
problem_name() (sage.numerical.backends.generic_backend.GenericBackend method), 154
problem_name() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 270
problem_name() (sage.numerical.backends.glpk_backend.GLPKBackend method), 191
problem_name() (sage.numerical.backends.interactivelp_backend.InteractiveLPBackend method), 170
problem_name() (sage.numerical.backends.logging_backend.LoggingBackend method), 227
problem_type() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 97

parent() (sage.numerical.backends.logging_backend.LoggingBackend method), 255
random_dictionary() (in module sage.numerical.interactive_simplex_method), 136
random_element() (sage.numerical.interactive_sim-
S

sage.numerical.backends.cvxopt_backend
module, 232

sage.numerical.backends.cvxopt_sdp_backend
module, 274

sage.numerical.backends.generic_backend
module, 143

sage.numerical.backends.generic_sdp_backend
module, 265

sage.numerical.backends.glpk_backend
module, 175

sage.numerical.backends.glpk_backend
module, 205

sage.numerical.backends.glpk_graph_backend
module, 208

sage.numerical.backends.interactive_backend
module, 162

sage.numerical.backends.logging_backend
module, 243

sage.numerical.backends.ppl_backend
module, 220

sage.numerical.gauss_legendre
module, 139

sage.numerical.interactive_simplex_method
module, 83

sage.numerical.knapsack
module, 1

sage.numerical.linear_functions
module, 53

sage.numerical.linear_tensor
module, 63

sage.numerical.linear_tensor_constraints
module, 69

sage.numerical.linear_tensor_element
module, 67

sage.numerical.mip
module, 7

sage.numerical.optimize
module, 73

sage.numerical.sdp
module, 39

save() (sage.numerical.backends.logging_backend.LoggingBackend method), 258

SDPSolverException, 42

SDPVariable (class in sage.numerical.sdp), 43

SemidefiniteProgram (class in sage.numerical.sdp), 44

set_binary() (sage.numerical.mip.MixedIntegerLinearProgram method), 30

set_col_stat() (sage.numerical.backends.glpk_backend.GLPKBackend method), 193

set_integer() (sage.numerical.mip.MixedIntegerLinearProgram method), 30

set_max() (sage.numerical.mip.MIPVariable method), 12

set_max() (sage.numerical.mip.MixedIntegerLinearProgram method), 31

set_min() (sage.numerical.mip.MIPVariable method), 12

set_min() (sage.numerical.mip.MixedIntegerLinearProgram method), 31

set_multiplication_symbol() (sage.numerical.linear_functions.LinearFunctionsParent_class method), 60

set_objective() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 239

set_objective() (sage.numerical.backends.generic_backend.GenericBackend method), 156

set_objective() (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method), 271

set_objective() (sage.numerical.backends.glpk_backend.GLPKBackend method), 194

set_objective() (sage.numerical.backends.interactive_backend.InteractiveLPBackend method), 172

set_objective() (sage.numerical.backends.logging_backend.LoggingBackend method), 259

set_objective() (sage.numerical.backends.ppl_backend.PPLBackend method), 229

set_objective() (sage.numerical.mip.MixedIntegerLinearProgram method), 32

set_objective() (sage.numerical.sdp.SemidefiniteProgram method), 49

set_problem_name() (sage.numerical.mip.MixedIntegerLinearProgram method), 32

set_problem_name() (sage.numerical.sdp.SemidefiniteProgram method), 49

set_real() (sage.numerical.mip.MixedIntegerLinearProgram method), 32

set_row_stat() (sage.numerical.backends.glpk_backend.GLPKBackend method), 194

set_sense() (sage.numerical.backends.cvxopt_backend.CVXOPTBackend method), 239
set_verbosity()

set_vertex_demand()

set_vertices_demand()
Index

standard_form() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 97

style() (in module sage.numerical.interactive_simplex_method), 137

subset_sum() (sage.numerical.knapsack.Superincreasing method), 4

sum() (sage.numerical.mip.MixedIntegerLinearProgram method), 36

sum() (sage.numerical.sdp.SemidefiniteProgram method), 52

Superincreasing (class in sage.numerical.knapsack), 2

T

tensor() (sage.numerical.linear_functions.LinearFunctionsParent_class method), 60

U

update() (sage.numerical.interactive_simplex_method.LPAbstractDictionary method), 120

update() (sage.numerical.interactive_simplex_method.LPDictionary method), 125

update() (sage.numerical.interactive_simplex_method.LPRevisedDictionary method), 135

V

values() (sage.numerical.mip.MIPVariable method), 13

values() (sage.numerical.sdp.SDPVariable method), 43

variable() (in module sage.numerical.interactive_simplex_method), 138

variable_lower_bound() (sage.numerical.backends cvxopt_backend.CVXOPTBackend method), 242

variable_lower_bound() (sage.numerical.backends.generic_backend.GenericBackend method), 158

variable_lower_bound() (sage.numerical.backends.glpk_backend.GLPKBackend method), 203

variable_lower_bound() (sage.numerical.backends.interactive lp_backend.InteractiveLPBackend method), 174

variable_lower_bound() (sage.numerical.backends.logging_backend.LoggingBackend method), 261

variable_lower_bound() (sage.numerical.backends.ppl_backend.PPLBackend method), 231

variable_types() (sage.numerical.interactive_simplex_method.InteractiveLPProblem method), 98

variable_upper_bound() (sage.numerical.backends cvxopt_backend.CVXOPTBackend method), 242

variable_upper_bound() (sage.numerical.backends.generic_backend.GenericBackend method), 159

variable_upper_bound() (sage.numerical.backends.glpk_backend.GLPKBackend method), 203

variable_upper_bound() (sage.numerical.backends.interactive lp_backend.InteractiveLPBackend method), 174

variable_upper_bound() (sage.numerical.backends.logging_backend.LoggingBackend method), 261

variable_upper_bound() (sage.numerical.backends.ppl_backend.PPLBackend method), 231

vertices() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 218

W

warm_up() (sage.numerical.backends.glpk_backend.GLPKBackend method), 204

write_ccdata() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 218

write_graph() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 219

write_lp() (sage.numerical.backends.generic_backend.GenericBackend method), 159

write_lp() (sage.numerical.backends.glpk_backend.GLPKBackend method), 204

write_lp() (sage.numerical.backends.logging_backend.LoggingBackend method), 262

write_lp() (sage.numerical.mip.MixedIntegerLinearProgram method), 37

write_maxflow() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 219

write_mincost() (sage.numerical.backends.glpk_graph_backend.GLPKGraphBackend method), 220

write_mps() (sage.numerical.backends.generic_backend.GenericBackend method), 159

write_mps() (sage.numerical.backends.glpk_backend.GLPKBackend method), 205

write_mps() (sage.numerical.backends.logging_backend.LoggingBackend method), 262

write_mps() (sage.numerical.mip.MixedIntegerLinearProgram method), 37
\textbf{X}  
\begin{itemize}
  \item \texttt{x() \ (sage.numerical.interactive_simplex_method.InteractiveLPProblem method)}, \texttt{98}
  \item \texttt{x_B() \ (sage.numerical.interactive_simplex_method.LPRevisedDictionary method)}, \texttt{135}
  \item \texttt{x_N() \ (sage.numerical.interactive_simplex_method.LPRevisedDictionary method)}, \texttt{135}
\end{itemize}

\textbf{Y}  
\begin{itemize}
  \item \texttt{y() \ (sage.numerical.interactive_simplex_method.LPRevisedDictionary method)}, \texttt{135}
\end{itemize}

\textbf{Z}  
\begin{itemize}
  \item \texttt{zero()} \ (sage.numerical.backends.generic_backend.GenericBackend method)}, \texttt{160}
  \item \texttt{zero()} \ (sage.numerical.backends.generic_sdp_backend.GenericSDPBackend method)}, \texttt{273}
  \item \texttt{zero()} \ (sage.numerical.backends.ppl_backend.PPLBackend method)}, \texttt{232}
\end{itemize}