University of Oxford

Master's thesis

Creating Semidefinite Programming Interface for Sagemath

Author:
Ingolfur Edvardsson

Supervisor:
Dmitrii Pasechnik

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the Department of Computer Science

2014

UNIVERSITY OF OXFORD

Abstract

Department of Computer Science

Master of Science

Creating Semidefinite Programming Interface for Sagemath

by Ingolfur Edvardsson

Sagemath (also known as Sage) is a large-scale open-source Python-based computer algebra/numerics toolbox which provides a seamless interface between various open-source computer algebra and numerics systems, such as GAP, Maxima, Singular, PARI/GP, numpy, scipy, etc. It is gaining in popularity in the research community as an open-source alternative for Mathematica, Maple, Magma, and Matlab. Semidefinite Programming (SDP) problems are a novel class of problems in convex optimisation, which properly contain such well-known problems as Linear Programming (LP), Convex Quadratic Programming, etc, and is a very powerful modelling tool in discrete optimisation, control theory, polynomial optimisation, etc.

The present MSc thesis seeks to explain the design and implementation of an SDP interface for Sage, which constituted most of my MSc project, and provide the necessary mathematical and CS background. The meta-aim was to create an interface which can deal with optimization problems with polynomial constraints, using SDP relaxations. An essential step was taken by this project in that direction by first writing an LP backend, fully capable of solving various LPs, and then designing and implementing an SDP interface. The final version of the SDP interface is capable of solving different kinds of SDPs using at least two independent solvers, which the user can choose. Future developers should also find it easy to add new backends for other SDP solvers.

Parts of the code written are already accepted into the main development branch of Sage, and will appear in the upcoming release; the remaining code will be merged there in due course, upon passing the review process.

Acknowledgements

The project was a great chance for learning and professional development and I could not have done it without help from several people. I first and foremost want to express my deepest gratitude to my advisor, Dmitrii Pasechnik, for his excellent guidance throughout the project. I also want to thank Nathann Cohen and the open-source community of Sage for their input as well. At last, I want to thank for the great support I received from many selfless employees here at University of Oxford. Without their support, my project would not have been possible.

Contents

List of Tables

Chapter 1

Introduction

1.1 Intoduction

The aim of the project is to design and implement a Semidefinite Programming (usually abbreviated SDP) interface for Sagemath [?], known as Sage, so that Sage can deal with optimization problems with polynomial constraints, using SDP relaxations. The project started with theoretical reading and preparation, followed by an implementation of a linear programming backend, cvxopt [?]. Finally the SDP interface was implemented along the cvxopt-sdp solver backend, thus making a sizeable contribution towards the final aim, as well as bringing new functionality.

1.2 The importance of an SDP interface

Nowadays, users who want to solve an SDP, using Sage, have to express the problem in a very restrictive standard form which is required by the solvers, rather than in a natural way that follows mathematical conventions. Also, the user would have to know how to formulate the problem for each solver. Not to mention the inefficient way of reading and writing to a file which requires the user to write a script—creating a lot of problems—for solvers not having a Python interface. Thus it is important to have an interface so the user doesn't have to have a comprehensive understanding on how each solver works. By creating an SDP interface, this can be achieved.

As far as numerics is concerned, Matlab [?] is without a doubt the most famous alternative to Sage. Matlab has few interfaces that are capable of solving an SDP. The two most prominent packages to solve an SDP using Matlab are `YALMIP` [?] and `CVX` [?].

1.3 Sagemath

Sage is a open-source mathematics software which builds on top of many existing open-source packages and libraries. They describe Sage as so:

> William Stein started the Sage project in 2004 and still leads the project. His frustration with proprietary mathematical software was his main motivation to create a viable open-source alternative. Just as Firefox is an alternative to Internet Explorer or OpenOffice.org is an alternative to Microsoft Office, Sage is a comprehensive open source alternative to Magma, Maple, Mathematica, and Matlab. Sage consists of a collection of mathematical software and a core library bundling the functionality of these components into one consistent experience. Additionally to that it provides a framework to express mathematical calculations and a library of mathematical algorithms. Mathematics is very old and encompasses many very different topics. It is hard to come up with one unique approach that suites beginners as well as experts. Sage tries to solve this and "is doing remarkably well at keeping a balance between ease-of-use for beginners and high-end users." as David Kohel once said [**?**].

Open-source gives the researcher a possibility to see the code behind the methods, which in many cases can be essential step in, for instance, a mathematical proof. This property, along with the fact that Sage uses a general-purpose popular programming language, Python (unlike Matlab, Mathematica, etc.) makes Sage an easy to use software and thus we focus on developing an easy to use semidefinite programming interface, as this is an essential and valuable by itself step towards the overall project aim.

1.4 Linear Programming

Linear programming (usually abbreviated LP) is a method to maximize or minimize certain value in a mathematical model whose requirements are represented by linear relationships. They can represented in the following form:

$$\text{Maximize: } c^\top x \tag{1.1}$$

$$\text{subject to: } Ax \leq b \tag{1.2}$$

$$\tag{1.3}$$

where x represents the vector of variables to be determined, c and b are vectors of known coefficients, A is a known matrix of coefficients, and the operator $(\cdot)^\top$ is the matrix transpose [**?**].

1.5 Semidefinite Programming

SDP is considered among the most powerful tools in the theory and practice of approximation algorithms and methods based on semidefinite programming have been very prominent in optimization since the 1990s. Semidefinite programs can be defined in an equational form:

$$\text{Maximize: } \sum_{i,j=1}^{n} c_{ij}x_{ij} \tag{1.4}$$

$$\text{Subject to: } \sum_{i,j=1}^{n} a_{ijk}x_{ij} = b_k, \qquad k = 1 \dots m \tag{1.5}$$

$$X \succeq 0 \tag{1.6}$$

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

The class of SDPs is one of the largest class of optimization problems that is possible to solve efficiently, in practice as well in theory. SDPs play an important role in many research areas, such as approximation algorithms, combinatorial optimization, computational complexity, graph theory, geometry, quantum computing and real algebraic geometry. Thus it is a topic of great interest [**?**].

SDP can be solved efficiently in theory and practice, that is, if one uses an interior point algorithm, such as the one CVXOPT provides. This means it can be solved in time polynomial in the size needed to code its description provided some natural non-degeneracy conditions hold for the input [**?**].

1.6 The CVXOPT Library

CVXOPT is a free software package for convex optimization based on the Python programming language under the GNU v3 licence. It can be integrated in other software via Python extension modules, like Sage for instance. The main purpose of the CVX-OPT is to make the development of software for convex optimization as straightforward as possible. It uses the fact that Python has a powerful standard library as well as Python being a high-level programming language. The current version of CVXOPT is at 1.1.7 (released in June 2014). However, we used the previous versions of CVXOPT, namely, version 1.1.6. According to the changelog, the difference between those two version should not affect the SDP or the LP solvers in any way [**?**].

1.6.1 Algorithm parameters for the CVXOPT solver

Here we have the list of algorithm control parameters that CVXOPT accepts. They are accessible via the dictionary solvers.options. The following parameters are possible to change:

`show_progress` True or False; turns the output to the screen on or off (default: True).

`maxiters` maximum number of iterations (default: 100).

`abstol` absolute accuracy (default: 1e-7).

`reltol` relative accuracy (default: 1e-6).

`feastol` tolerance for feasibility conditions (default: 1e-7).

`refinement` number of iterative refinement steps when solving KKT equations (default: 0 if the problem has no second-order cone or matrix inequality constraints; 1 otherwise) [**?**].

1.6.2 Possible outcomes

After calling any of the methods provided by the `cvxopt.solvers`, the solver returns a dictionary *d* which can tell us if the solver successfully managed to approximate a solution. The four possible outcomes are `'optimal'`, which tells us if the solver successfully managed to come up with a solution. `'primal infeasible'` or `'dual infeasible'` tells us that approximate solution wasn't found and finally `'unknown'` means the algorithm terminated early due to numerical difficulties or because the maximum number of iterations was reached [**?**].

Chapter 2

The CVXOPT LP Backend

2.1 Overview

SDP is an extension of LP since every LP can be expressed as an SDP. Hence it was wise to design and implement an LP backend solvers first. We created a backend for the LP solver which was based on the open-source cvxopt solver. Not only was it important for the community to have access to this solver but part of the code could be used for the Semidefinite Programming part. Addition of this backend increases the number of different LP solver backends available in Sage to six.

2.2 Solving linear Programs in Sage

Sage is capable of solving many different kinds of mathematical problems, in particular, LPs. We assume user wants to solve the following problem:

$$\begin{aligned} \text{Max: } & x + y + 3z && (2.1)\\ \text{Such that: } & x + 2y \leq 4 && (2.2)\\ & 5z - y \leq 8 && (2.3)\\ & x, y, z \geq 0 && (2.4) \end{aligned}$$

To solve it, we need to instantiate the MILP class (stands for MixedIntegerLinearProgram) to create an object p, use p to create 3 variables x, y and z, set the objective, add the constraints and finally call the solve method. The code is mostly self-explanatory:

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

```
sage: v = p.new_variable(real=True, nonnegative=True)
sage: x, y, z = v['x'], v['y'], v['z']
sage: p.set_objective(x + y + 3*z)
sage: p.add_constraint(x + 2*y <= 4)
sage: p.add_constraint(5*z - y <= 8)
sage: p.solve()
8.8
```

which means that $x + y - 3z$ is at maximum 8.8 for the constraint defined above.

Notice that when we create the object p we pick a solver by setting the *solver* argument. In this case we picked the GLPK solver. However, our goal was to write a backend for the CVXOPT solver so one can use the CVXOPT solver to solve the linear program by defining the MILP object, p, in the following way:

```
sage: p = MixedIntegerLinearProgram(solver="CVXOPT")
```

To create a CVXOPT backend it is important to understand how to use the CVXOPT solver.

2.3 LP example using CVXOPT

Let's take an example on how the CVXOPT LP solver works. Let's say we have the following linear program:

$$\begin{aligned}
\text{Minimize: } & -4x_1 - 5x_2 && (2.5)\\
\text{Subject to: } & 2x_1 + x_2 \leq 3 && (2.6)\\
& x_1 + 2x_2 \leq 3 && (2.7)\\
& x_1 \geq 0, \quad x_2 \geq 0 && (2.8)
\end{aligned}$$

Assuming we have set up the CVXOPT library the correct way, we can use the LP solver of the CVXOPT library to solve the problem above in the following way:

```
>>> from cvxopt import matrix, solvers
>>> c = matrix([-4., -5.])
>>> G = matrix([[2., 1., -1., 0.], [1., 2., 0., -1.]])
>>> h = matrix([3., 3., 0., 0.])
>>> sol = solvers.lp(c, G, h)
>>> print(sol['x'])
[ 1.00e+00]
[ 1.00e+00]
```

This means the expression $-4x_1 - 5x_2$ takes the lowest value when $x_1 \approx 1$ and $x_2 \approx 1$ so we get $-4(1) + -5(1) \approx -9$ [?].

2.4 The importance of the CVXOPT solver

Sage already has 5 LP solvers backends implemented. However, they are all based on the Simplex method, making them very different from the CVXOPT solver which is based on the interior point method.

2.4.1 The difference between the Simplex method and interior point method

The Simplex method is based on the fact that the objective function's maximum or minimum must occur in an extreme point of the region cut out by the linear constraints. The Simplex method is a remarkable algorithm and was for instance chosen as one of the top 10 algorithm of the twentieth century by the journal Computing in Science and Engineering [?].

From the beginning of 1990s until now, interior point algorithms have dominated the research on convex optimization problems. Their popularity is mostly based on the fact they reach a high accurate solution in a small number of iterations (from 10 to 50), almost independent of problem size and type [?].

The main difference between those two methods is that the interior point method, which CVXOPT is based on, runs in polynomial time [?], while the Simplex method, which the rest of the solvers backends are based on, are not know to be able to do so. However, in practice, some LP problems are solved significantly faster with one type of solver. Thus it is important for the user to be able to select between solvers of both types, making the CVXOPT solver very important in practice since it is the only solver in Sage based on interior point method [?].

Another important difference between those two methods is that the interior point based solvers return an interior point of the optimal face as an optimal solution, rather than returning a vertex solution. This often gives a more desirable solution since sometimes the user wants to average over the optimal face. The following example should give an idea what we mean.

Lets say we have the following linear program:

$$\begin{aligned} \text{Max: } & z && (2.9) \\ \text{Such that: } & x \leq 100 && (2.10) \\ & y \leq 100 && (2.11) \\ & z \leq 100 && (2.12) \\ & x, y, z \geq 0 && (2.13) \end{aligned}$$

The reader should find it pretty obvious that the solution is of course $z = 100$. However, what happens to x and y? Since the Simplex based solvers pick a vertex solution, it simply leaves the x and y as 0. However, the interior point solvers, such as CVXOPT, gives an interior point of the optimal face and therefore the x and y get the value 50 which might be a more meaningful solution since it is the average of 100 and 0, that is $\frac{(100-0)}{2} = 50$. The example would look something like this when solved using Sage. First we use the default GLPK solver, which is a Simplex method based solver:

```
sage: p = MixedIntegerLinearProgram(solver = "GLPK")
sage: x = p.new_variable(nonnegative=True)
sage: p.set_objective(x[2])
sage: #we create a constraint cube
sage: p.add_constraint(x[0] <= 100)
sage: p.add_constraint(x[1] <= 100)
sage: p.add_constraint(x[2] <= 100)
sage: round(p.solve(),2)
100.0
sage: round(p.get_values(x[0]),2)
0.0
sage: round(p.get_values(x[1]),2)
0.0
```

As we expected, we get 0.0 in both cases. Let us run the same code. However, this time we pick CVXOPT as the solver. We get:

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

```
sage: p.set_objective(x[2])
sage: #we create a constraint cube
sage: p.add_constraint(x[0] <= 100)
sage: p.add_constraint(x[1] <= 100)
sage: p.add_constraint(x[2] <= 100)
sage: round(p.solve(),2)
100.0
sage: round(p.get_values(x[0]),2)
50.0
sage: round(p.get_values(x[1]),2)
50.0
```

Here the x and the y get the value 50.0 as expected.

2.5 Parameters for the `cvxopt.solvers.lp` method

The `cvxopt.solvers.lp(c, G, h[, A, b[, solver[, primalstart[, dualstart]]]])` is the method that solves the linear program from Section **??**. As we can tell from the declaration, there are three parameters we must provide c, G and h. First one is c which is the coefficients of the objective function. The next one is G which is the coefficients of the matrix A in Section **??** and at last the h is the b in the same section.

2.6 Important classes for the CVXOPT backend

Before we begin describing the implementation of the CVXOPT backend, it is important to have a good understanding on some of the classes the CVXOPT backend talks to. There are two classes worth to mention. The *MixedIntegerLinearProgram* (MILP) class and the *GenericBackend* class.

2.6.1 The MixedIntegerLinearProgram class

Let's look at how the LP interface is implemented in Sage.

The *MixedIntegerLinearProgram* (MILP) class (the LP interface class) is approximately 2300 lines where each method is fully documented and unit tested. The basic usage of the class is described in section **??**. It's importnant to understand how it works since it

Table 2.1: MixedIntegerLinearProgram's methods table

Function name	Description of its functionality
`add_constraint`	Adds a constraint to the “MixedIntegerLinearProgram“
`base_ring`	Return the base ring
`constraints`	Returns a list of constraints, as 3-tuples
`get_backend`	Returns the backend instance used
`get_max`	Returns the maximum value of a variable
`get_min`	Returns the minimum value of a variable
`get_values`	Return values found by the previous call to “solve()“
`is_binary`	Tests whether the variable is a binary
`is_integer`	Tests whether the variable is an integer
`is_real`	Tests whether the variable is a real
`linear_constraints_parent`	Return the parent for all linear constraints
`linear_function`	Construct a new linear function
`linear_functions_parent`	Return the parent for all linear functions
`new_variable`	Returns an instance of “MIPVariable“ associated
`number_of_constraints`	Returns the number of constraints assigned so far
`number_of_variables`	Returns the number of variables used so far
`polyhedron`	Returns the polyhedron defined by the Linear Program
`remove_constraint`	Removes a constraint from self
`remove_constraints`	Remove several constraints
`set_binary`	Sets a variable or a “MIPVariable“ as binary
`set_integer`	Sets a variable or a “MIPVariable“ as integer
`set_max`	Sets the maximum value of a variable
`set_min`	Sets the minimum value of a variable
`set_objective`	Sets the objective of the “MixedIntegerLinearProgram“
`set_problem_name`	Sets the name of the “MixedIntegerLinearProgram“
`set_real`	Sets a variable or a “MIPVariable“ as real
`show`	Prints “MixedIntegerLinearProgram“ in a human-readable
`solve`	Solves the “MixedIntegerLinearProgram“
`solver_parameter`	Return or define a solver parameter
`sum`	Sums the sequence of LinearFunction elements
`write_lp`	Write the linear program as a LP file
`write_mps`	Write the linear program as a MPS file

is responsible for passing the linear program, provided by the user, to the backend to be solved. Table **??** lists all the function and a small description of each of the function.

It is important to understand the structure of the MILP class in order to write a working backend that talks properly to the MILP class.

Table 2.2: GenericBackend's / CVXOPT's methods table

Function name	Description of its functionality
`add_variable`	Adds a new variable
`add_variables`	Adds several new variables
`set_variable_type`	Defines the variable's type (binary, integer or cont.)
`objective_coefficient`	Sets or gets the coefficient of a variable in the objective
`set_objective`	Sets the whole objective
`set_verbosity`	Sets the log level
`add_linear_constraint`	Adds a linear constraint
`add_col`	Adds a column to the matrix
`add_linear_constraints`	Adds several linear constraints
`solve`	Solves the problem
`get_objective_value`	Returns the value of the objective function
`get_variable_value`	Returns the value of a variable given by the solver.
`ncols`	Returns the number of columns
`nrows`	Returns the number of rows
`is_maximization`	Returns true if we are maximizing the objective function
`problem_name`	Returns or defines the problem's name
`row`	Returns a row from the matrix by index
`row_bounds`	Returns a pair (lower_bound,upper_bound) for a row
`column_bounds`	Returns a pair (lower_bound,upper_bound) for a column
`is_variable_binary`	Returns if True if variable is a binary
`is_variable_integer`	Returns if True if variable is an integer
`is_variable_continuous`	Returns if True if variable is continuous
`row_name`	Returns the name of the row called
`column_name`	Returns the name of the column called
`variable_upper_bound`	Returns or define the upper bound on a variable
`variable_lower_bound`	Returns or define the lower bound on a variable
`solver_parameter`	Returns or define a solver parameter

2.6.2 The GenericBackend class

The GenericBackend is a parent class to all linear solver backends. The MILP class creates an instance of the GenericBackend. Since all the backends inherits the GenericBackend, the GenericBackend serves as an interface, that is, it basically lists the methods that should be defined by an LP Solver, in our case the CVXOPT solver. Almost all the methods of the GenericBackend immediately raise "NotImplementedError" exceptions when called, and are obviously meant to be replaced by the solver-specific method. Let us look at table **??** for a list of all the methods that return "NotImplementedError". Remember, since the CVXOPT backend will inherit the GenericBackend, this is also a list of all the methods we need to implement for the CVXOPT backend.

To be clear, when we talk about "the matrix", columns or rows, we are referring to the matrix A in Chapter **??**.

It is worth to mention that The GenericBackend has the method

```
GenericBackend get_solver(constraint_generation = False, solver = None)
```

which is used by the MILP class to get a instance of the solver being used.

2.7 The Implementation of the CVXOPT LP backend

The final implementation of the CVXOPT backend was one class, exactly 999 lines, fully documented and unit tested, capable of solving a linear program as described in chapter **??**.

The MILP class (see table **??**) requires you to define each variable as binary, integer or real. Since the CVXOPT library only operates on real numbers, we implemented the methods for CVXOPT backend with that in mind. For instance, `is_variable_binary()` simply returns False since we can assume that all the variables for CVXOPT backend operates on real numbers.

2.7.1 Cython

The class was written in Cython, the same language as the other backends were written in. Cython combines the power of C and Python and lets you write Python code that calls natively back and forth from and to C code. Not only does Cython therefore extend Python by making it possible to call C functions, without the need for a lot of boilerplate code, but we can also utilize the efficiency of C when we need to make some of the more demanding computation as fast as possible. This is vital when writing a solver's backend where time is of the essence [**?**].

2.7.2 Class variables and the constructor

The following code shows the class variables as well as their initializations in the constructor of the CVXOPT backend:

```
self.objective_function = [] #c_matrix in the example for cvxopt
self.G_matrix = []
self.prob_name = None
self.obj_constant_term = 0
self.is_maximize = 1
```

```
        self.row_lower_bound = []
        self.row_upper_bound = []
        self.col_lower_bound = []
        self.col_upper_bound = []

        self.row_name_var = []
        self.col_name_var = []

        self.param = {"show_progress":False,
                      "maxiters":100,
                      "abstol":1e-7,
                      "reltol":1e-6,
                      "feastol":1e-7,
                      "refinement":0 }

        if maximization:
            self.set_sense(+1)
        else:
            self.set_sense(-1)
```

Notice that the `G_matrix` is basically the A matrix from chapter **??** with the $-b$ matrix appended. The `obj_constant_term` is the constant term for the objective_function and all the col and row bounds are used to bound constraints and variables. Notice however that they can be implemented in the `G_matrix` variable and were mostly added for convenience.

At last, we can see we added the default values for the algorithm parameters, as described in Chapter **??**.

2.7.3 The implementation of several major methods

The CVXOPT backend consisted of 27 methods as can be seen in Table **??** and here are few of the more complicated methods shown.

2.7.3.1 The implementation of `add_linear_constraint`

Let us look at the code for the method `add_linear_constraint()`

```
cpdef add_linear_constraint(self, coefficients, lower_bound,
 upper_bound, name=None):
    for c in coefficients:
        while c[0] > len(self.G_matrix)-1:
            self.add_variable()
    for i in range(len(self.G_matrix)):
        self.G_matrix[i].append(0.0)
    for c in coefficients:
        self.G_matrix[c[0]][-1] = c[1]

    self.row_lower_bound.append(lower_bound)
    self.row_upper_bound.append(upper_bound)
    self.row_name_var.append(name)
```

First of all, it was important to understand how the MILP class sent the list of coefficients c. It came as pairs of (i, v) where i was the index and v was the value of the coefficient for that variable. First we add as many variables as needed. Then we create a zero row in `G_matrix` that is a constraint that only consists of coefficients with value 0.0. Finally we add the relevant value from c to the corresponding row in the `G_matrix`.

2.7.3.2 The implementation of `solve()`

Let us look at the code for the method `solve()`

```
    cpdef int solve(self) except -1:
        from cvxopt import matrix, solvers
        h = []

        #for the equation bounds
        for eq_index in range(self.nrows()):
            h.append(self.row_upper_bound[eq_index])
            #upper bound is already in G
            if self.row_lower_bound[eq_index] != None:
                h.append(-1 * self.row_lower_bound[eq_index])
                for cindex in range(len(self.G_matrix)):
                    if cindex == eq_index:
                        # after multiplying the eq by -1
                        self.G_matrix[cindex].append(-1)
                    else:
```

```
                    self.G_matrix[cindex].append(0)




        #for the upper bounds (if there are any)
        for i in range(len(self.col_upper_bound)):
            if self.col_upper_bound[i] != None:
                h.append(self.col_upper_bound[i])
                for cindex in range(len(self.G_matrix)):
                    if cindex == i:
                        self.G_matrix[cindex].append(1)
                    else:
                        self.G_matrix[cindex].append(0)
            if self.col_lower_bound[i] != None:
                h.append(self.col_lower_bound[i])
                for cindex in range(len(self.G_matrix)):
                    if cindex == i:
                        # after multiplying the eq by -1
                        self.G_matrix[cindex].append(-1)
                    else:
                        self.G_matrix[cindex].append(0)

        G = []
        for col in self.G_matrix:
            tempcol = []
            for i in range(len(col)):
                tempcol.append( float(col[i]) )
            G.append(tempcol)

        G = matrix(G)

        #cvxopt minimizes on default
        if self.is_maximize:
            c = [-1 * float(e) for e in self.objective_function]
        else:
            c = [float(e) for e in self.objective_function]
        c = matrix(c)

        h = [float(e) for e in h]
        h = matrix(h)

        #solvers comes from the cvxopt library
        for k,v in self.param.iteritems():
            solvers.options[k] = v
        self.answer = solvers.lp(c,G,h)

        #possible outcomes
        if self.answer['status'] == 'optimized':
            pass
        elif self.answer['status'] == 'primal infeasible':
            raise MIPSolverException("CVXOPT: primal infeasible")
        elif self.answer['status'] == 'dual infeasible':
            raise MIPSolverException("CVXOPT: dual infeasible")
        elif self.answer['status'] == 'unknown':
            raise MIPSolverException("CVXOPT: Terminated early due to \
             numerical difficulties or because the maximum number of \
             iterations was reached.")
        return 0
```

The `solve()` method is definitely the most important method of the CVXOPT class. The purpose of this method is to first, take the values from the bounds and add it to the `G_matrix` as well as to the h variable which represents the h in chapter **??**. Next we go through the `G_matrix`, convert it to float type which the matrix from cvxopt package can understand, and then convert the list G to G of type cvxopt.matrix. We multiply the elements of c by -1 if we want to maximize since the cvxopt minimizes the objectives by default. Finally we add the solve parameters, call the `solvers.lp` from the cvxopt package and handle the errors as shown above (see chapter **??**).

2.7.3.3 The implementation of `get_objective_value`

First let us look at the code for the method `get_objective_value()`

```
    cpdef get_objective_value(self):
        sum = self.obj_constant_term
        i = 0
        for v in self.objective_function:
            sum += v * float(self.answer['x'][i])
            i+=1
        return sum
```

Here we calculate the `objetive_value` by going through the values, already calculated by the solve method, multiply it with the coefficient of the `objective_function` and add it to the sum. This is the method `mip.solve()` calls to get the desired value.

2.7.3.4 The implementation and design of `ncols()` and `nrows()`

We decided not to use the `G_matrix` variable to define the `ncols()` and `nrows()` since we will run into troubles when `G_matrix` is small or empty. `ncols()` was therefore implemented by computing the length of the `objective_function` list since every time we call `add_variable()` in the backend, we add an integer (the coefficient) to the `objective_function` list (worst case scenario, if that variable is not part of the objective function, we add 0.0 to that list. Thus `ncols()` returns the correct value.

The `nrow()` was simply implemented by computing the number of elements in the `row_upper_bounds` list.

2.7.4 Documentation and Testing

Each and every method in the CVXOPT backend was thoroughly tested and documented. The class passed all the 202 unit test written and contained 720 lines of unit tests, examples and documentation in total. These tests greatly improve the stability of the product and ensured that the code meets its design and behaves as intended.

2.8 Results

As was mention above, the CVXOPT backend final version was a 999 line linear program solver, capable of solving various different kind of linear programs. The backend was designed and implemented under strict object oriented principles, fully tested and documented.

2.8.1 Benchmark

To make sure this backend wouldn't slow down the solving process a lot, we ran some benchmarks using the built-in %%timeit function. We took a very simple linear program to be solved and here is the results.

```
sage: %%timeit -p 4
sage: p = MixedIntegerLinearProgram(solver="cvxopt")
sage: v = p.new_variable(real=True, nonnegative=True)
sage: x, y, z = v['x'], v['y'], v['z']
sage: p.set_objective(x + y + 3*z)
sage: p.add_constraint(x + 2*y <= 4)
sage: p.add_constraint(5*z - y <= 8)
sage: round(p.solve(), 2)
1000 loops, best of 3: 1.556 ms per loop
```

and when we ran the same code using only the cvxopt library to solve the same problem, we got:

```
%%timeit -p 4
sage: from cvxopt import matrix, solvers
sage: c = matrix([-1., -1.,-3.])
sage: G = matrix([[1., 0.,-1.,0.,0.], [2., -1.,0.,-1.,0.],[0.,5.,0.,0.,-1.]])
sage: h = matrix([4., 8.,0.,0.,0.])
sage: sol = solvers.lp(c, G, h)
sage: float(sol['x'][0])+float(sol['x'][1])+3*float(sol['x'][2])
1000 loops, best of 3: 1.232 ms per loop
```

As we can see, the one using the backend was $1.556 - 1.232 = 0.324$ms slower than using the cvxopt library straight away. Hence, using the backend to solve a small LP slows down the process by approximately $\frac{1.556-1.232}{1.232} \approx 26.3\%$. That is totally acceptable and expected since we need to recreate the `G_matrix` inside the `cvxopt_backend.solve()` method.

2.8.2 Current status in the open-source community

After writing the backend, a ticked was created in the Sage developer community portal (running `trac` server, an integrated SCM and project management tool) where the code was pushed. After several reviews, modifications, fixes and improvements on the backend, it got a positive review. The 26th of June, the ticket was closed. We can thus expect to see the backend in the next release of Sage, version 6.4 [**?**].

Chapter 3

The SDP Interface

3.1 Overview

For the past fifteen years, SDP has become one of the most exciting developments in mathematical programming. SDP has applications in many different fields traditional convex constrained optimization, control theory, and combinatorial optimization. SDP can be solved using interior-point methods such as the one CVXOPT provides and usually requires about the same amount of computational resources as linear optimization. Hence they can usually be solved fairly efficiently in practice as well as in theory [?].

Since solving SDP has become such an important topic in the past few years, we believe that creating an easy to use SDP interface with at least one solver in an open-source environment like Sage (**??**), is a key property for both the users who want to solve certain SDP as well as for other software developers and researchers who want to add new solvers to an unified interface.

3.2 The design of the SDP interface

As mention before, the main goal is to design and implement an SDP interface with one solver, capable of solving an SDP.

The first step was to understand how SDP works in general. The courses Foundation of Computer Science and Probability and Computing help tremendously to understand advanced topics in Linear Optimization, such as relaxation of a problem, the well known MAX-CUT problem (covered in Probability and Computing), the relevant complexity classes (to know what an SDP solver is capable of and what to avoid) and other topics. This made, for instance, the reading of [?] feasible.

Next step was to design the process on how the SDP interface should work. Let us give a practical example how we want to solve an SDP (see Chapter **??** for the definition of SDPs).

Imagine you want to minimize $x_0 - x_1$ where:

$$\begin{bmatrix}1 & 2\\ 2 & 3\end{bmatrix} x_0 + \begin{bmatrix}3 & 4\\ 4 & 5\end{bmatrix} x_1 \preceq \begin{bmatrix}5 & 6\\ 6 & 7\end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} x_0 + \begin{bmatrix}2 & 2\\ 2 & 2\end{bmatrix} x_1 \preceq \begin{bmatrix}3 & 3\\ 3 & 3\end{bmatrix} \tag{3.2}$$

where $\forall x_i \in \mathbb{R}^+$. To solve the SDP above, we decided this would be the process:

- You have to create an instance of `SemidefiniteProgram`. We add the parameter `maximization=False` since we want to minimize $x_0 - x_1$.
- Create an dictionary x of integer variables x via $x =$ `p.new_variable()`.
- Add those two inequalities as inequality constraints via `add_constraint` (`<sage.numerical.sdp.SemidefiniteProgram.add_constraint>`.)
- Specify the objective function via `set_objective` (`<sage.numerical.sdp.SemidefiniteProgram.set_objective>`). In our case it is $x_0 - x_1$. If it is a pure constraint satisfaction problem, specify it as `None`.
- To check if everything is set up correctly, you can print the problem via `show` (`<sage.numerical.sdp.SemidefiniteProgram.show>`.)
- And finally we call `Solve` (`<sage.numerical.sdp.SemidefiniteProgram.solve>`) to solve the SDP and print the solution.

The following example shows the desired behaviour:

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

Table 3.1: the SDP methods table

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

```
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: print 'Objective Value:', round(p.solve(),3)
Objective Value: -3.0
sage: p.show()
Minimization:
  x_0 - x_1
Constraints:
  constraint_0: [1.0 2.0][2.0 3.0]x_0 + [3.0 4.0][4.0 5.0]x_1 <=
       \ [5.0 6.0][6.0 7.0]
  constraint_1: [1.0 1.0][1.0 1.0]x_0 + [2.0 2.0][2.0 2.0]x_1 <=
       \ [3.0 3.0][3.0 3.0]
Variables:
   x_0,  x_1
```

To accomplish this, we came to the conclusion that we needed to implement the methods listed in Table ??. The table consists of 15 methods which make it possible for the user to solve an SDP in an easy and efficient way.

3.3 Solving SDP example using CVXOPT

First we look at how the method `cvxopt.solvers.sdp` takes in its parameters and later we look at an example on how to use the solver to solve a well defined SDP problem.

3.3.1 Parameters for the `cvxopt.solvers.sdp` method

The `cvxopt.solvers.sdp(c[, Gl, hl[, Gs, hs[, A, b[, solver[, primalstart[, dualstart]]]]]])` is the method that solves the SDP from chapter **??**. The essential parameters we provide are three in total, the c, Gs and hs. We also took advantage of the parameter `solver` but more on that later.

The first parameter, c, is a list of the coefficients of the objective function. The Gs is a list which has n sub-lists, where n stands for the number of constraints for the SDP. Each sub-list is a flattened out list of type `cvxopt.matrix` and stands for a single matrix coefficient. The hs is the b_k in section **??**.

3.3.2 Example

Let us make an example on how the CVXOPT SDP solver works. Let us say we have the following SDP:

Minimize: $x_1 - x_2 + x_3$

Subject to:

$$x_1 \begin{bmatrix} -7 & -11 \\ -11 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 7 & -18 \\ -18 & 8 \end{bmatrix} + x_3 \begin{bmatrix} -2 & -8 \\ -8 & 1 \end{bmatrix} \preceq \begin{bmatrix} 33 & -9 \\ -9 & 26 \end{bmatrix}$$

$$x_1 \begin{bmatrix} 21 & -11 & 0 \\ -11 & 10 & 8 \\ 0 & 8 & 5 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 10 & 16 \\ 10 & -10 & -10 \\ 16 & -10 & 3 \end{bmatrix} + x_3 \begin{bmatrix} -5 & 2 & -17 \\ 2 & -6 & 8 \\ -17 & 8 & 6 \end{bmatrix} \preceq \begin{bmatrix} 14 & 9 & 40 \\ 9 & 91 & 10 \\ 40 & 10 & 15 \end{bmatrix}$$

Assuming we have access to the CVXOPT library, we can use the SDP solver of the CVXOPT library to solve this problem above in the following way:

```
>>> from cvxopt import matrix, solvers
>>> c = matrix([1.,-1.,1.])
>>> G = [ matrix([[-7., -11., -11., 3.],
                  [ 7., -18., -18., 8.],
                  [-2.,  -8.,  -8., 1.]]) ]
```

```
>>> G += [ matrix([[-21., -11.,   0., -11.,  10.,   8.,   0.,   8., 5.],
                   [  0.,  10.,  16.,  10., -10., -10.,  16., -10., 3.],
                   [ -5.,   2., -17.,   2.,  -6.,   8., -17.,  8., 6.]]) ]
>>> h = [ matrix([[33., -9.], [-9., 26.]]) ]
>>> h += [ matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]]) ]
>>> sol = solvers.sdp(c, Gs=G, hs=h)
>>> print(sol['x'])
[-3.68e-01]
[ 1.90e+00]
[-8.88e-01]
```

This means the expression $x_1 - x_2 + x_3$ takes the lowest value when $x_1 \approx -0.368$, $x_2 \approx 1.90$ and $x_3 \approx -0.888$ so we get $-0.368 - 1.90 - 0.888 \approx -3.156$ [?].

3.4 The implementation of the SDP interface

The final implementation consisted of 4 classes, `SemidefiniteProgram`, `GenericSDPBackend`, `CVXOPTSDPBackend` and `DSDPSDPBackend`. (We also borrowed code from the ticket `Add a matrix of constraints in a LP` [?]). The four classes we wrote were 2706 lines in total.

3.4.1 The `SemidefiniteProgram` class

The `SemidefiniteProgram` class was written in Cython (see subsection ??) and when finished, was exactly 1342 lines of code. The code was fully documented with examples, each method was unit tested and comments were added as needed. The class was written with table ?? in mind.

The `SemidefiniteProgram` class was designed to be a link between Sage, SDP and semidefinite programming solvers. A Semidefinite Programming (SDP) consists of SDP variables, linear tensors on these variables, and an objective function which is to be maximised or minimised under certain constraints.

3.4.1.1 The SDPVariable

It was necessary to create a small helper class for the `SemidefiniteProgram` class. We wrote a class `SDPVariable` which was to be used in our linear tensor functions. Each instance of `SDPVariable` was linked with one instance of the `SemidefiniteProgram`. It also had a variable `_name` but more importantly, a variable `_dict` which kept the values of the elements a single `SDPVariable` was capable of creating. This class was written so after an `SDPVariable` has been created, and we call an element of that variable, it doesn't matter if it exists or not. Let us look at the following example:

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

This shows that the user can call $v[0]$ without initiate it.

Another very important functionality is the fact we can use matrices as coefficients for a linear function that consists of sum of SDPVariables. This makes it possible to write, for example:

```
sage: a = matrix([[1,2],[2,3]])
sage: b = matrix([[3,4],[4,5]])
sage: p = SemidefiniteProgram()
sage: v = p.new_variable()
sage: a*v[0] + b*v[1]
[x_0 + 3*x_1   2*x_0 + 4*x_1]
[2*x_0 + 4*x_1 3*x_0 + 5*x_1]
sage: type(a*v[0] + b*v[1])
<type 'sage.numerical.linear_tensor_element.LinearTensor'>
```

This makes the interface much more user friendly and helps keeping the process as simple as possible.

3.4.1.2 The SDPSolverException

A `SDPSolverException` was also created to handle different outputs from the solvers. Most solvers give some sort of information on how well it did in solving a SDP problem. If the solver doesn't manage to solve a particular SDP, the solver backend is responsible for throwing an `SDPSolverException` and to give information on what went wrong. For instance, the possible outcomes for the CVXOPT solver are listed in sub-section **??**.

3.4.1.3 Class variables and the constructor

Since the class was designed to be a link between Sage, SDP and semidefinite programming solvers, the most important parameter for the constructor was the name of the solver which was to be used to solve the SDP.

The class has several class variables. However, the most important one was definitely the `_backend` variable which was of type `GenericSDPBackend` (described in **??**.) This variable is used over and over again to talk to the SDP solver backend.

Let us now look at the implementation of several important methods.

3.4.1.4 Implementation of `add_constraint`

Here we see the code for the method `add_constraint` in the class `SemidefiniteProgram`:

```
def add_constraint(self, linear_function, name=None):
    if linear_function is 0:
        return

    from sage.numerical.linear_tensor_constraints import \
      is_LinearTensorConstraint
    from sage.numerical.linear_tensor import is_LinearTensor

    if is_LinearTensorConstraint(linear_function) or\
      is_LinearConstraint(linear_function):
        c = linear_function
        if c.is_equation():
            self.add_constraint(c.lhs()-c.rhs(), name=name)
            self.add_constraint(-c.lhs()+c.rhs(), name=name)
        else:
            self.add_constraint(c.lhs()-c.rhs(), name=name)

    elif is_LinearFunction(linear_function) or \
      is_LinearTensor(linear_function):
        l = linear_function.dict().items()
        l.sort()
        self._backend.add_linear_constraint(l, name)

    else:
        raise ValueError('argument must be a linear function or \
         constraint, got '+str(linear_function))
```

The `add_constraint` takes in an instance of a linear tensor equation (written `==`) or inequalities (written `<=` or `>=`) where the variables are of type `SDPVariable` and the coefficients are matrices. An example of this is `a*x[0] <= b` where a and b are matrices and x is a variable created with the `SemidefiniteProgram.new_variable` method.

This function was implemented as a recursive one. If we have an equation, we look at it as two inequalities. In both cases (either the linear tensor constraint is equation or inequality) we move the right hand side to the left hand side to create an instance of a LinearTensor (not LinearTensorConstraint as it was before). Then we create a dictionary and send it to the backend. If we send something else than a LinearTensor (or LinearFunction, to handle some special cases) to `add_constraint`, we raise a `ValueError`.

Notice that we call the `l.sort()` to make sure it is possible, when implementing a new backend, to iterate through the constrains without worrying about if they come in the right order or not. We can assume the constraint are few enough for the `sort()` method not to slow down the process.

3.4.1.5 Implementation of `solve`

The solve method inside `SemidefiniteProgram` serves as a wrapper function, calling first the solve method of the backend, and then returning the objective value computed. The code simply became

```
def solve(self):
    self._backend.solve()
    return self._backend.get_objective_value()
```

3.4.1.6 Documentation and Testing

Each and every method in `SemidefiniteProgram` was thoroughly tested and documented. The class passed all the 222 unit test written and contained 741 lines of unit tests, examples and documentation in total. These tests greatly improve the stability of the product and ensured that the code meets its design gaols and behaves as intended.

3.4.2 The GenericSDPBackend class

The GenericSDPBackend class was written to serve as a superclass for all the backends. That is, when we implement a backend, it should inherit the GenericSDPBackend. This class had to be created since then we can let SemidefiniteProgram class have a variable _backend of type GenericSDPBackend which simplifies the code dramatically.

This class lists the methods that should be defined by a backend with an SDP Solver. Most of the methods immediately raise NotImplementedError exceptions when called, and are obviously meant to be replaced by the solver-specific method. However, there are two methods implemented in the class. First it is the default_sdp_solver which returns or sets the default SDP solver and then get_solver which returns the appropriate backend. This class makes it easier to implement more SDP backends in the future.

3.4.3 The CVXOPTSDPBackend class

After we finished writing the SemidefiniteProgram and GenericSDPBackend classes, it was important to have at least one backend solver so these classes could be used in practice. We then decided to implement a backend for the SDP solver, provided by CVXOPT library (see section ??). The class CVXOPTSDPBackend was based on the CVXOPTBackend from chapter ??. This backend has the same list of methods. However, we had to reimplement most of the methods to make it handle the fact the coefficients were matrices and not real numbers.

One example of this is the add_linear_constraint method. Let us look at the code:

```
cpdef add_linear_constraint(self, coefficients, name=None):
    from sage.matrix.matrix import is_Matrix
    for t in coefficients:
        m = t[1]
        if not is_Matrix(m):
            raise Exception("The coefficients must be matrices")
        if not m.is_square():
            raise Exception("The matrix has to be a square")
        if self.matrices_dim.get(self.nrows()) != None and \
         m.dimensions()[0] != self.matrices_dim.get(self.nrows()):
            raise Exception("The matrces have to be of the same dimension")
    self.coeffs_matrix.append(coefficients)
    self.matrices_dim[self.nrows()] = m.dimensions()[0] #
```

```
        self.row_name_var.append(name)
```

First notice that some error handling is being done to make sure the matrices are as expected. This was important to make sure we couldn't add a broken constraint to the constraint matrix. After that we append the coefficients to the variable `coeffs_matrix` which contains all constraints.

3.4.3.1 The `solve()` method

As mention before (see Table ??), the backend contains the solve method, responsible for solving the SDP. This function is in some sense similar to the `solve()` for the Linear CVXOPT backend, described in ??. One major difference though is that we utilize the fact CVXOPT offers to use more than its default solver. Therefore we get an additional solver for almost free by adding this code to the `solve()` method:

```
...
if self.solver == "DSDP":
    self.answer = solvers.sdp(c,Gs=G_matrix,hs=h_matrix,solver="dsdp")
else:
    self.answer = solvers.sdp(c,Gs=G_matrix,hs=h_matrix)
...
```

This will be explained in more detail in Subsection ??.

3.4.3.2 Documentation and Testing

All the methods were thoroughly tested and documented. The class passed all the 147 unit test written and it contained 441 lines of unit tests, examples and documentation in total.

3.4.4 The `DSDPSDPBackend` class

As mentioned in ??, we took advantage of the fact that the default CVXOPT SDP solver can easily be changed to an external one called DSDP. This can be accomplished by putting a solver parameter into the solve function described in Subsection ??. This gives the user the ability to choose among two solvers which can be essential in practice as well as in academic research.

3.4.4.1 The DSDP solver

The DSDP solver is as well an interior-point solver and can in some cases be more appropriate to use than the CVXOPT solver. The DSDP version 5 was released in 2005 and is described by the author in the following way:

> Initiated in 1997, DSDP has developed into an efficient and robust general-purpose solver for semidefinite programming. Its features include a convergence proof with polynomially bounded worst-case complexity, primal and dual feasible solutions when they exist, certificates of infeasibility when solutions do not exist, initial points that can be feasible or infeasible, relatively low memory requirements for an interior-point method, sparse and low- rank data structures, extensibility that allows applications to customize the solver and improve its performance, a subroutine library that enables it to be linked to larger applications, scalable performance for large problems on parallel architectures, and a well-documented interface and examples of its use. The package has been used in many applications and tested for efficiency, robustness, and ease of use [**?**].

The DSDP solver is however, not a part of Sage. Thus it is important that the CVXOPT library is built with a DSDP support as well as the DSDP will be added to Sage.

3.4.4.2 The DSDP backend

Due to the fact we implemented the CVXOPT with the possibility to switch between solvers, we only needed to implement one method in the `DSDPSDPBackend` which returns a modified version of the CVXOPT backend. Therefore it is sufficient to have:

```
    def return_modified_cvxopt(self):
        return CVXOPTSDPBackend(solver = "dsdp")
```

and then the `GenericSDPBackend` calls this method when the user asks for the DSDP solver.

To use the DSDP solver to solve an SDP, the user would follow the same routine as in section **??**. However, we would change

```
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization = False)
...
```

to

```
sage: p = SemidefiniteProgram(solver = "dsdp", maximization = False)
...
```

This is the ideal way for the user to switch between solvers.

3.5 Results

The final version of the SDP interface was, as mention above, 4 fully tested classes, capable of solving various different kinds of SDP with at least two independent solvers. Future developers should find it easy to add new backends for other SDP solvers since the interface was clearly built with a good separation between the interface and the backends.

We believe all the problems we encountered on the way were solved. Not only did we implement an easy to use SDP interface as the goal was, but we also made it so the user now has the freedom to choose between two different solvers.

3.5.1 Benchmark

Again we ran a benchmark using the `%%timeit` function, to make sure neither the interface nor the backend would slow down the solving process a lot. We chose a simple SDP to be solved and here are the results:

```
sage: %%timeit -p 4
sage: p = SemidefiniteProgram(solver = "cvxopt", maximization=False)
sage: x = p.new_variable()
sage: p.set_objective(x[0] - x[1] + x[2])
sage: a1 = matrix([[-7., -11.], [-11., 3.]])
sage: a2 = matrix([[7., -18.], [-18., 8.]])
sage: a3 = matrix([[-2., -8.], [-8., 1.]])
sage: a4 = matrix([[33., -9.], [-9., 26.]])
sage: b1 = matrix([[-21., -11., 0.], [-11., 10., 8.], [0.,   8., 5.]])
sage: b2 = matrix([[0.,  10.,  16.], [10., -10., -10.], [16., -10., 3.]])
sage: b3 = matrix([[-5.,   2., -17.], [2.,  -6.,   8.], [-17.,  8., 6.]])
sage: b4 = matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]])
sage: p.add_constraint(a1*x[0] + a2*x[1] + a3*x[2] <= a4)
```

```
sage: p.add_constraint(b1*x[0] + b2*x[1] + b3*x[2] <= b4)
sage: p.solve()
100 loops, best of 3: 5.604 ms per loop
```

and when we ran the same code using only the cvxopt library to solve the same problem, we got:

```
sage: %%timeit -p 4
sage: from cvxopt import matrix, solvers
sage: solvers.options["show_progress"] = False
sage: c = matrix([1.,-1.,1.])
sage: G = [ matrix([[-7., -11., -11., 3.],
                   [ 7., -18., -18., 8.],
                   [-2.,  -8.,  -8., 1.]]) ]
sage: G += [ matrix([[-21., -11.,   0., -11.,  10.,   8.,   0.,   8., 5.],
                    [  0.,  10.,  16.,  10., -10., -10.,  16., -10., 3.],
                    [ -5.,   2., -17.,   2.,  -6.,   8., -17.,   8., 6.]]) ]
sage: h = [ matrix([[33., -9.], [-9., 26.]]) ]
sage: h += [ matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]]) ]
sage: sol = solvers.sdp(c, Gs=G, hs=h)
sage: float(sol['x'][0])-float(sol['x'][1])+float(sol['x'][2])
100 loops, best of 3: 4.524 ms per loop
```

We can see that when we use the backend, we are $5.604 - 4.524 = 1.080$ms slower than using the cvxopt library directly. Thus, when we are solving a small SDP, it slows down the process by approximately $\frac{5.604-4.524}{4.524} \approx 23.9\%$. Again, this is more than acceptable and expected since extra computation is needed when using the SDP interface.

Notice however that when we pick a larger SDP, the percentage should get closer to zero since then the main effort goes into solving the actual SDP instead of formulating it so the backend understands it.

3.5.2 Current status in the open-source community

When the implementation was done and the interface, the backends and other classes passed all their unit tests, a ticket was created in the Sage developer community portal. The code is currently being reviewed and is under discussion by the members of the community [**?**].

3.5.3 Future work

We have now taken a major step towards dealing with optimization problems with polynomial constraints by creating an SDP interface for Sage. It would be interesting to use the SDP interface to create a module for sum-of-squares calculations as described here [**?**].

Having access to more backends would also be helpful if the user wants to try out other solvers. Creating backend for another modern solver such as CSDP [**?**] would then give the user more freedom and might speed up the computation time for certain SDP problems.

References

[1] #16490 (create a linear programming backend for cvxopt) – sage. http://trac.sagemath.org/ticket/16490. [Online; accessed 28-August-2014].

[2] #16714 (add a matrix of constraints in a lp) – sage. http://trac.sagemath.org/ticket/16714. [Online; accessed 29-August-2014].

[3] #16929 (creating sdp interface) – sage. http://trac.sagemath.org/ticket/16929. [Online; accessed 04-September-2014].

[4] Cone programming — cvxopt user's guide. http://cvxopt.org/userguide/coneprog.html#algorithm-parameters. [Online; accessed 18-August-2014].

[5] Cone programming — cvxopt user's guide. http://cvxopt.org/userguide/coneprog.html. [Online; accessed 14-August-2014].

[6] Cython: C-extensions for Python. http://http://cython.org/. [Online; accessed 16-August-2014].

[7] Guest editors' introduction: The top 10 algorithms. http://www.computer.org/csdl/mags/cs/2000/01/c1022.html. [Online; accessed 13-August-2014].

[8] Home — cvxopt. http://cvxopt.org. [Online; accessed 14-August-2014].

[9] Interior-point methods for large-scale cone programming. http://www.seas.ucla.edu/~vandenbe/publications/mlbook.pdf. [Online; accessed 13-August-2014].

[10] Linear programming - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Linear_programming. [Online; accessed 11-August-2014].

[11] MATLAB - The Language of Technical Computing - MathWorks United Kingdom. http://www.mathworks.co.uk/products/matlab/. [Online; accessed 3-September-2014].

[12] Sage - press - about. http://www.sagemath.org/library-press.html. [Online; accessed 10-August-2014].

[13] Semidefinite programming mode — CVX Users' Guide. http://web.cvxr.com/cvx/doc/sdp.html. [Online; accessed 3-September-2014].

[14] Semidefinite programming — cvxopt user's guide. http://cvxopt.org/userguide/coneprog.html#semidefinite-programming. [Online; accessed 22-August-2014].

[15] YALMIP Wiki Tutorials/Semidefinite Programming. http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Tutorials.SemidefiniteProgramming. [Online; accessed 3-September-2014].

[16] Steven J. Benson and Yinyu Ye. DSDP5: Software for semidefinite programming. Technical Report ANL/MCS-P1289-0905, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2005. Submitted to ACM Transactions on Mathematical Software, available http://www.mcs.anl.gov/hs/software/DSDP/DSDP5-P1289-0905.pdf.

[17] Brian Borchers. Csdp 2.3 user's guide. *Optimization Methods and Software*, 11(1-4):597–611, 1999.

[18] Bernd Gärtner and Jiri Matousek. *Approximation Algorithms and Semidefinite Programming*. Springer, 2012 edition, 1 2012.

[19] Camile WJ Hol and Carsten W Scherer. Sum of squares relaxations for polynomial semidefinite programming. In *Proc. Symp. on Mathematical Theory of Networks and Systems (MTNS), Leuven, Belgium*, 2004.

[20] Mark Iwen. Introduction to semidefinite programming (sdp). http://www.math.msu.edu/~markiwen/Teaching/MTH995/Papers/SDP_notes_Marina_Epelman_UM.pdf. [Online; accessed 22-August-2014].

[21] Jean-Bernard Lasserre. *Moments, Positive Polynomials and Their Applications (Imperial College Press Optimization Series)*. Imperial College Press, 1 edition, 10 2009.

[22] Daniel Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 296–305. ACM, 2001.