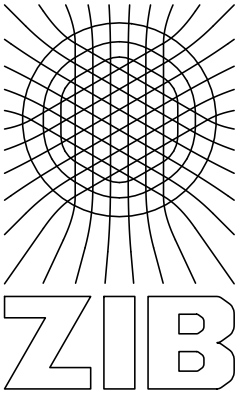
Konrad-Zuse-Zentrum
für Informationstechnik Berlin



Takustraße 7
D-14195 Berlin-Dahlem
Germany

GUILLAUME SAGNOL

Picos Documentation

Release 0.1.1

ZIB-Report 12-48 (December 2012)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Picos Documentation

Release 0.1.1

Guillaume Sagnol

December 10, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

PICOS is a user friendly interface to several conic and integer programming solvers, very much like YALMIP under MATLAB.

The main motivation for PICOS is to have the possibility to enter an optimization problem as a *high level model*, and to be able to solve it with several *different solvers*. Multidimensional and matrix variables are handled in a natural fashion, which makes it painless to formulate a SDP or a SOCP. This is very useful for educational purposes, and to quickly implement some models and test their validity on simple examples.

Furthermore, with PICOS you can take advantage of the python programming language to read and write data, construct a list of constraints by using python list comprehensions, take slices of multidimensional variables, etc.

It must also be said that PICOS is only a unified interface to other already existing interfaces of optimization solvers. So you have to install some additional packages each time you want to use PICOS with a new solver (see *a list of supported solvers*, and the *packages you will have to install* to use them). Furthermore, since PICOS is just another interface layer, one should expect an overhead due to PICOS in the solution time.

Here is a very simple example of the usage of PICOS:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',1, vtype='integer') #scalar integer variable
>>> prob.add_constraint(x<5.2)                    #x less or equal to 5.2
>>> prob.set_objective('max',x)                   #maximize x
>>> print prob
---------------------
optimization problem (MIP):
1 variables, 1 affine constraints
x   : (1, 1), integer
    maximize x
such that
  x < 5.2
---------------------
>>> sol = prob.solve(solver='zibopt',verbose=0) #solve using the ZIB optimization suite
>>> print x                                     #optimal value of x
5.0
```

Currently, PICOS can handle the following class of optimzation problems. A list of currently interfaced solvers can be found *here*.

- Linear Programming (**LP**)
- Mixed Integer Programming (**MIP**)
- Convex Quadratically constrained Quadratic Programming (**convex QCQP**)
- Second Order Cone Programming (**SOCP**)
- Semidefinite Programming (**SDP**)
- General Quadratically constrained Quadratic Programming (**QCQP**)
- Mixed Integer Quadratic Programming (**MIQP**)

There exists a number of similar projects, so we provide a (non-exhausive) list below, explaining their main differences with PICOS:

- CVXPY:

  This is a python interface that can be used to solve any convex optimization problem. However, CVXPY interfaces only the open source solver cvxopt for disciplined convex programming (**DCP**)

- Numberjack:

  This python package also provides an interface to the integer programming solver scip, as well as satisfiability (**SAT**) and constraint programming solvers (**CP**).

- OpenOpt:

  This is probably the most complete optimization suite written in python, handling a lot of problem types and interfacing many opensource and commercial solvers. However, the user has to transform every optimization problem into a canonical form himself, and this is what we want to avoid with PICOS.

- puLP:

  A user-friendly interface to a bunch of **LP** and **MIP** solvers.

- PYOMO:

  A modelling language for optimization problems, *a la* AMPL.

- pyOpt:

  A user-friendly package to formulate and solve general nonlinear constrained optimization problems. Several open-source and commercial solvers are interfaced.

- python-zibopt:

  This is a user-friendly interface to the ZIB optimization suite for solving mixed integer programs (**MIP**). PICOS provides an interface to this interface.

1.1 First Example

We give below a simple example of the use of PICOS, to solve an SOCP which arises in *optimal experimental design*. More examples can be found *here*. Given some observation matrices $A_1, \ldots, A_s$, with $A_i \in \mathbb{R}^{m \times l_i}$, and a coefficient matrix $K \in \mathbb{R}^{m \times r}$, the problem to solve is:

$$\begin{array}{ll} \underset{\substack{\mu \in \mathbb{R}^s \\ \forall i \in [s],\ Z_i \in \mathbb{R}^{l_i \times r}}}{\text{minimize}} & \displaystyle\sum_{i=1}^{s} \mu_i \\ \text{subject to} & \displaystyle\sum_{i=1}^{s} A_i Z_i = K \\ & \forall i \in [s],\ \|Z_i\|_F \leq \mu_i, \end{array}$$

where $\|M\|_F := \sqrt{\text{trace} M M^T}$ denotes the Frobenius norm of M. This problem can be entered and solved as follows with PICOS:

```
import picos as pic
import cvxopt as cvx

#generate data
A = [   cvx.sparse([[1 ,2 ,0 ],
                    [2 ,0 ,0 ]]),
        cvx.sparse([[0 ,2 ,2 ]]),
        cvx.sparse([[0 ,2 ,-1],
                    [-1,0 ,2 ],
```

```
                    [0 ,1 ,0 ]])
    ]
K = cvx.sparse([[1 ,1 ,1 ],
                [1 ,-5,-5]])

#size of the data
s = len(A)
m = A[0].size[0]
l = [ Ai.size[1] for Ai in A ]
r = K.size[1]

#creates a problem and the optimization variables
prob = pic.Problem()
mu = prob.add_variable('mu',s)
Z  = [prob.add_variable('Z[' + str(i) + ']', (l[i],r))
      for i in range(s)]

#convert the constants into params of the problem
A = pic.new_param('A',A)
K = pic.new_param('K',K)

#add the constraints
prob.add_constraint( pic.sum([ A[i]*Z[i] for i in range(s)], #summands
                             'i',                            #name of the index
                             '[s]'                           #set to which the index
                            ) == K                           #belongs
                   )
prob.add_list_of_constraints( [ abs(Z[i]) < mu[i] for i in range(s)], #constraints
                               'i',                                   #index of the constraints
                               '[s]'                                  #set to which the index belongs
                             )

#sets the objective
prob.set_objective('min', 1 | mu ) # scalar product of the vector of all ones with mu

#display the problem
print prob

#call to the solver cvxopt
sol = prob.solve(solver='cvxopt', verbose = 0)

#show the value of the optimal variable
print '\n  mu ='
print mu

#show the dual variable of the equality constraint
print'\nThe optimal dual variable of the'
print prob.get_constraint(0)
print 'is :'
print prob.get_constraint(0).dual
```

This generates the output:

```
---------------------
optimization problem  (SOCP):
15 variables, 6 affine constraints, 15 vars in 3 SO cones

mu  : (3, 1), continuous
Z   : list of 3 variables, different sizes, continuous

    minimize ⟨ |1| | mu ⟩
such that
  Σ_{i in [s]} A[i]*Z[i] = K
```

```
  ||Z[i]|| < mu[i] for all i in [s]
---------------------

  mu =
[ 6.60e-01]
[ 2.42e+00]
[ 1.64e-01]


The optimal dual variable of the
# (3x2)-affine constraint : Σ_{i in [s]} A[i]*Z[i] = K #
is :
[-3.41e-01]
[ 9.16e-02]
[-1.88e-01]
[-3.52e-01]
[ 2.32e-01]
[ 2.59e-01]
```

1.2 Solvers

Below is a list of the solvers currently interfaced by PICOS. We have indicated the classes of optimization problems that the solver can handle via PICOS. Note however that the solvers listed below might have other features that are *not handled by PICOS*.

- cvxopt (LP, SOCP, SDP, GP)
- smcp (LP, SOCP, SDP)
- mosek (LP, MIP, (MI)SOCP, convex QCQP, MIQP)
- cplex (LP, MIP, (MI)SOCP, convex QCQP, MIQP)
- gurobi (LP, MIP, (MI)SOCP, convex QCQP, MIQP)
- zibopt (soplex + scip : LP, MIP, MIQP, general QCQP).

To use one of these solver, make sure that the python interface to this solver is correctly installed and linked in your `PYTHONPATH` variable. The sites of the solvers give instructions to do this, except for *zibopt*, for which you must install a separate interface: python-zibopt. To check your installation, you can simply verify that `import cvxopt` (resp. `smcp`, `mosek`, `cplex`, `zibopt`, `gurobipy`) does not raise an `ImportError`. The command

```
>>> import picos;picos.tools.available_solvers()
```

returns the list of correctly installed solvers.

1.3 Requirements

PICOS has two dependencies: numpy and cvxopt. (cvxopt is needed even if you do not use the cvxopt solvers, because picos relies on the `sparse matrices` defined in cvxopt.)

In addition, you must install separately the python interfaces to each *solver* you want to use.

1.4 Installation

After having downloaded the latest version of picos, and extracted it in the directory of your choice, you can install it by typing the following line as root in a terminal:

```
$ python setup.py install
```

If you do not have administrator rights, you can also do a local installation of picos with the *prefix scheme*. For example:

```
$ python setup.py install --prefix ~/python
```

and make sure that `$HOME'/python/lib/python2.x/site-packages/'` is in your `PYTHONPATH` variable.

To test your installation, you can run the test file:

```
$ python picos/test_picos.py
```

This will generate a table with a list of results for each available solver and class of optimization problems.

1.5 License

1.6 Author and contributors

- Author: Picos initial author and current primary developer is:

  Guillaume Sagnol, <sagnol(a t)zib.de>

- Contributors: People who contributed to Picos and their contributions (in no particular order) are:
  - Bertrand Omont
  - Elmar Swarat

CHAPTER

TWO

TUTORIAL

First of all, let us import the PICOS module and cvxopt

```
>>> import picos as pic
>>> import cvxopt as cvx
```

We now generate some arbitrary data, that we will use in this tutorial.

```
>>> pairs = [(0,2), (1,4), (1,3), (3,2), (0,4),(2,4)]   #a list of pairs
>>> A = []
>>> b = ( [0 ,2 ,0 ,3 ],                                 #a tuple of 5 lists,
...       [1 ,1 ,0 ,5 ],                                 #each of length 4
...       [-1,0 ,2 ,4 ],
...       [0 ,0 ,-2,-1],
...       [1 ,1 ,0 ,0 ]
...     )
>>> for i in range(5):
...     A.append(cvx.matrix(range(i-3,i+5),(2,4)))     #A is a list of 2x4 matrices
>>> D={'Peter': 12,
...    'Bob'  : 4,
...    'Betty': 7,
...    'Elisa': 14
...    }
```

Let us now create an instance P of an optimization problem

```
>>> prob = pic.Problem()     #create a Problem instance
```

2.1 Variables

We will now create the variables of our optimization problem. This is done by calling the method `add_variable()`. This function adds an instance of the class `Variable` in the dictionary `prob.variables`, and returns a reference to the freshly added variable. As we will next see, we can use this `Variable` to form affine and quadratic expressions.

```
>>> t = prob.add_variable('t',1) #a scalar
>>> x = prob.add_variable('x',4) #a column vector
>>> Y = prob.add_variable('Y',(2,4)) #a matrix
>>> Z = []
>>> for i in range(5):
...     Z.append( prob.add_variable('Z[{0}]'.format(i),(4,2))  )# a list of 5 matrices
>>> w={}
>>> for p in pairs:   #a dictionary of (scalar) binary variables, indexed by our pairs
...     w[p] = prob.add_variable('w[{0}]'.format(p),1 , vtype='binary')
```

Now, if we try to display a variable, here is what we get:

```
>>> w[2,4]
# variable w[(2, 4)]:(1 x 1),binary #
>>> Y
# variable Y:(2 x 4),continuous #
```

Also note the use of the attributes `name`, `value`, `size`, and `vtype`:

```
>>> w[2,4].vtype
'binary'
>>> x.vtype
'continuous'
>>> x.vtype='integer'
>>> x
# variable x:(4 x 1),integer #
>>> x.size
(4, 1)
>>> Z[1].value = A[0].T
>>> Z[0].is_valued()
False
>>> Z[1].is_valued()
True
>>> Z[2].name
'Z[2]'
```

2.2 Affine Expressions

We will now use our variables to create some affine expressions, which are stored as instance of the class `AffinExp`, and will be the core to define an optimization problem. Most python operators have been overloaded to work with instances of `AffinExp` (a list of available overloaded operators can be found in the doc of `AffinExp`). For example, you can form the sum of two variables by writing:

```
>>> Z[0]+Z[3]
# (4 x 2)-affine expression: Z[0] + Z[3] #
```

The transposition of an affine expression is done by appending `.T`:

```
>>> x
# variable x:(4 x 1),integer #
>>> x.T
# (1 x 4)-affine expression: x.T #
```

2.2.1 Parameters as constant affine expressions

It is also possible to form affine expressions by using parameters stored in data structures such as a `list` or a `cvxopt matrix` (In fact, any type that is recognizable by the function `_retrieve_matrix()`).

```
>>> x + b[0]
# (4 x 1)-affine expression: x + [ 4 x 1 MAT ] #
>>> x.T + b[0]
# (1 x 4)-affine expression: x.T + [ 1 x 4 MAT ] #
>>> A[0] * Z[0] + A[4] * Z[4]
# (2 x 2)-affine expression: [ 2 x 4 MAT ]*Z[0] + [ 2 x 4 MAT ]*Z[4] #
```

In the above example, you see that the list `b[0]` was correctly converted into a 4×1 vector in the first expression, and into a 1×4 vector in the second one. This is because the overloaded operators always try to convert the data into matrices of the appropriate size.

If you want to have better-looking string representations of your affine expressions, you will need to convert the parameters into constant affine expressions. This can be done thanks to the function `new_param()`:

```
>>> A = pic.new_param('A',A)              #list of constant affine expressions
...                                       #  [A[0],...,A[4]]
>>> b = pic.new_param('b',b)              #list of constant affine expressions
...                                       #  [b[0],...,b[4]]
>>> D = pic.new_param('D',D)              #dictionary of constant AffExpr,
...                                       #        indexed by 'Peter', 'Bob', ...
>>> alpha = pic.new_param('alpha',12)     #a scalar parameter

>>> alpha
# (1 x 1)-affine expression: alpha #
>>> D['Betty']
# (1 x 1)-affine expression: D[Betty] #
>>> b
[# (4 x 1)-affine expression: b[0] #,
 # (4 x 1)-affine expression: b[1] #,
 # (4 x 1)-affine expression: b[2] #,
 # (4 x 1)-affine expression: b[3] #,
 # (4 x 1)-affine expression: b[4] #]
>>> print b[0]
[ 0.00e+00]
[ 2.00e+00]
[ 0.00e+00]
[ 3.00e+00]
```

The above example also illustrates that when a *valued* affine expression `exp` is printed, it is its value that is displayed. For a non-valued affine expression, **__repr__** and **__str__** produce the same result, a string of the form `'# (size)-affine expression:  string-representation #'`. Note that the constant affine expressions, as `b[0]` in the above example, are always *valued*. To assign a value to a non-constant `AffinExp`, you must set the `value` property of every variable involved in the affine expression.

```
>>> x_minus_1 = x - 1                   #note that 1 is recognized as the
>>> x_minus_1                           #(4x1)-vector with all ones
# (4 x 1)-affine expression: x -|1| #
>>> print x_minus_1
# (4 x 1)-affine expression: x -|1| #
>>> x_minus_1.is_valued()
False
>>> x.value = [0,1,2,-1]
>>> x_minus_1.is_valued()
True
>>> print x_minus_1
[-1.00e+00]
[ 0.00e+00]
[ 1.00e+00]
[-2.00e+00]
```

We also point out that `new_param()` converts lists into vectors and lists of lists into matrices (given in row major order). In contrast, tuples are converted into list of affine expressions:

```
>>> pic.new_param('vect',[1,2,3])                      # a vector of dimension 3
# (3 x 1)-affine expression: vect #
>>> pic.new_param('mat',[[1,2,3],[4,5,6]])             # a (2x3)-matrix
# (2 x 3)-affine expression: mat #
>>> pic.new_param('list_of_scalars',(1,2,3))           # a list of 3 scalar parameters
[# (1 x 1)-affine expression: list_of_scalars[0] #,
 # (1 x 1)-affine expression: list_of_scalars[1] #,
 # (1 x 1)-affine expression: list_of_scalars[2] #]
>>> pic.new_param('list_of_vectors',([1,2,3],[4,5,6]))  # a list of 2 vector parameters
[# (3 x 1)-affine expression: list_of_vectors[0] #,
 # (3 x 1)-affine expression: list_of_vectors[1] #]
```

2.2.2 Overloaded operators

OK, so now we have some variables (`t`, `x`, `w`, `Y`, and `Z`) and some parameters (`A`, `b`, `D` and `alpha`). Let us create some affine expressions with them.

```
>>> A[0] * Z[0]                              #left multiplication
# (2 x 2)-affine expression: A[0]*Z[0] #
>>> Z[0] * A[0]                              #right multiplication
# (4 x 4)-affine expression: Z[0]*A[0] #
>>> A[1] * Z[0] * A[2]                       #left and right multiplication
# (2 x 4)-affine expression: A[1]*Z[0]*A[2] #
>>> alpha*Y                                  #scalar multiplication
# (2 x 4)-affine expression: alpha*Y #
>>> t/b[1][3] - D['Bob']                     #division by a scalar and substraction
# (1 x 1)-affine expression: t / b[1][3] -D[Bob] #
>>> ( b[2] | x )                             #dot product
# (1 x 1)-affine expression: ⟨ b[2] | x ⟩ #
>>> ( A[3] | Y )                             #generalized dot product for matrices:
                                             #    (A|B)=trace(A*B.T)
# (1 x 1)-affine expression: ⟨ A[3] | Y ⟩ #
```

We can also take some subelements of affine expressions, by using the standard syntax of python slices:

```
>>> b[1][1:3]                                #2d and 3rd elements of b[1]
# (2 x 1)-affine expression: b[1][1:3] #
>>> Y[1,:]                                   #2d row of Y
# (1 x 4)-affine expression: Y[1,:] #
>>> x[-1]                                    #last element of x
# (1 x 1)-affine expression: x[-1] #
>>> A[2][:,1:3]*Y[:,-2::-2]                  #extended slicing with (negative) steps
# (2 x 2)-affine expression: A[2][:,1:3]*( Y[:,-2::-2] ) #
```

In the last example, we keep only the second and third columns of `A[2]`, and the columns of `Y` with an even index, considered in the reverse order. To concatenate affine expressions, the operators `//` and `&` have been overloaded:

```
>>> (b[1] & b[2] & x & A[0].T*A[0]*x) // x.T #vertical (//) and horizontal (&) concat.
# (5 x 4)-affine expression: [b[1],b[2],x,A[0].T*A[0]*x;x.T] #
```

When a scalar is added/substracted to a matrix or a vector, we interprete it as an elementwise addition of the scalar to every element of the matrix or vector.

```
>>> 5*x - alpha
# (4 x 1)-affine expression: 5*x + |-alpha| #
```

> **Warning:** Note that the string representation `'|-alpha|'` does not stand for the absolute value of `-alpha`, but for the vector whose all terms are `-alpha`.

2.2.3 Summing Affine Expressions

You can take the advantage of python syntax to create sums of affine expressions:

```
>>> sum([A[i]*Z[i] for i in range(5)])
# (2 x 2)-affine expression: A[0]*Z[0] + A[1]*Z[1] + A[2]*Z[2] + A[3]*Z[3] + A[4]*Z[4] #
```

This works, but you might have very long string representations if there are a lot of summands. So you'd better use the function `picos.sum()`):

```
>>> pic.sum([A[i]*Z[i] for i in range(5)],'i','[5]')
# (2 x 2)-affine expression: Σ_{i in [5]} A[i]*Z[i] #
```

It is also possible to sum over several indices

```
>>> pic.sum([A[i][1,j] + b[j].T*Z[i] for i in range(5) for j in range(4)],
...         ['i','j'],'[5]x[4]')
# (1 x 2)-affine expression: Σ_{i,j in [5]x[4]} |A[i][1,j]| + b[j].T*Z[i] #
```

A more complicated example, given in two variants: in the first one, `p` is a tuple index representing a pair, while in the second case we explicitely say that the pairs are of the form `(p0,p1)`:

```
>>> pic.sum([w[p]*b[p[1]-1][p[0]] for p in pairs],('p',2),'pairs')
# (1 x 1)-affine expression: Σ_{p in pairs} w[p]*b[p__1-1][p__0] #
>>> pic.sum([w[p0,p1]*b[p1-1][p0] for (p0,p1) in pairs],['p0','p1'],'pairs')
# (1 x 1)-affine expression: Σ_{p0,p1 in pairs} w[(p0, p1)]*b[p1-1][p0] #
```

It is also possible to sum over string indices (*see the documentation of* `sum()`):

```
>>> pic.sum([D[name] for name in D],'name','people_list')
# (1 x 1)-affine expression: Σ_{name in people_list} D[name] #
```

2.2.4 Objective function

The objective function of the problem can be defined with the function `set_objective()`. Its first argument should be `'max'`, `'min'` or `'find'` (*for feasibility problems*), and the second argument should be a scalar expression:

```
>>> prob.set_objective('max',( A[0] | Y )-t)
>>> print prob
---------------------
optimization problem (MIP):
59 variables, 0 affine constraints

w   : dict of 6 variables, (1, 1), binary
Z   : list of 5 variables, (4, 2), continuous
t   : (1, 1), continuous
Y   : (2, 4), continuous
x   : (4, 1), integer

    maximize ⟨ A[0] | Y ⟩ -t
such that
  []
---------------------
```

With this example, you see what happens when a problem is printed: the list of optimization variables is displayed, then the objective function and finally a list of constraints (in the case above, there is no constraint).

2.3 Norm of an affine Expression

The norm of an affine expression is an overload of the `abs()` function. If `x` is an affine expression, `abs(x)` is its Euclidean norm $\sqrt{x^T x}$.

```
>>> abs(x)
# norm of a (4 x 1)- expression: ||x|| #
```

In the case where the affine expression is a matrix, `abs()` returns its Frobenius norm, defined as $\|M\|_F := \sqrt{\text{trace}(M^T M)}$.

```
>>> abs(Z[1]-2*A[0].T)
# norm of a (4 x 2)- expression: ||Z[1] -2*A[0].T|| #
```

Note that the absolute value of a scalar expression is stored as a norm:

```
>>> abs(t)
# norm of a (1 x 1)- expression: ||t|| #
```

However, a scalar constraint of the form $|a^T x + b| \leq c^T x + d$ is handled as two linear constraints by PICOS, and so a problem with the latter constraint can be solved even if you do not have a SOCP solver available. Besides, note that the string representation of an absolute value uses the double bar notation. (Recall that the single bar notation `|t|` is used to denote the vector whose all values are `t`).

2.4 Quadratic Expressions

Quadratic expressions can be formed in several ways:

```
>>> t**2 - x[1]*x[2] + 2*t - alpha                    #sum of linear and
...                                                   #quadratic terms
#quadratic expression: t**2 -x[1]*x[2] + 2.0*t -alpha #
>>> (x[1]-2) * (t+4)                                  #product of two AffExpr
#quadratic expression: ( x[1] -2.0 )*( t + 4.0 ) #
>>> Y[0,:]*x                                          #Row vector multiplied
...                                                   #by column vector
#quadratic expression: Y[0,:]*x #
>>> (x +2 | Z[1][:,1])                               #dot product of 2 AffExpr
#quadratic expression: ⟨ x + |2.0| | Z[1][:,1] ⟩ #
>>> abs(x)**2                                         #recall that abs(x) is
...                                                   #the euclidean norm of x
#quadratic expression: ||x||**2 #
>>> (t & alpha) * A[1] * x                            #quadratic form
#quadratic expression: [t,alpha]*A[1]*x #
```

It is not possible (yet) to make a multidimensional quadratic expression.

2.5 Constraints

A constraint takes the form of two expressions separated by a relation operator.

2.5.1 Linear (in)equalities

Linear (in)equalities are understood elementwise. **The strict operators < and > denote weak inequalities** (*less or equal than* and *larger or equal than*). For example:

```
>>> (1|x) < 2                                         #sum of the x[i] < or = than 2
# (1x1)-affine constraint: ⟨ |1| | x ⟩ < 2.0 #
>>> Z[0] * A[0] > b[1]*b[2].T                         #A 4x4-elementwise inequality
# (4x4)-affine constraint: Z[0]*A[0] > b[1]*b[2].T #
>>> pic.sum([A[i]*Z[i] for i in range(5)],'i','[5]') == 0  #A 2x2 equality;
...                                                   #The RHS is the all-zero matrix.
# (2x2)-affine constraint: Σ_{i in [5]} A[i]*Z[i] = |0| #
```

Constraints can be added in the problem with the function `add_constraint()`:

```
>>> for i in range(1,5):
...       prob.add_constraint(Z[i]==Z[i-1]+Y.T)
>>> print prob
---------------------
optimization problem (MIP):
59 variables, 32 affine constraints

w   : dict of 6 variables, (1, 1), binary
```

```
Z   : list of 5 variables, (4, 2), continuous
t   : (1, 1), continuous
Y   : (2, 4), continuous
x   : (4, 1), integer

    maximize ⟨ A[0] | Y ⟩ -t
such that
  Z[1] = Z[0] + Y.T
  Z[2] = Z[1] + Y.T
  Z[3] = Z[2] + Y.T
  Z[4] = Z[3] + Y.T
---------------------
```

The constraints of the problem can then be accessed with the function `get_constraint()`:

```
>>> prob.get_constraint(2)                      #constraints are numbered from 0
# (4x2)-affine constraint: Z[3] = Z[2] + Y.T #
```

An alternative is to pass the constraint with the option `ret = True`, which has the effect to return a reference to the constraint you want to add. In particular, this reference can be useful to access the optimal dual variable of the constraint, once the problem will have been solved.

```
>>> mycons = prob.add_constraint(Z[4]+Z[0] == Y.T, ret = True)
>>> print mycons
# (4x2)-affine constraint : Z[4] + Z[0] = Y.T #
```

2.5.2 Groupping constraints

In order to have a more compact string representation of the problem, it is advised to use the function `add_list_of_constraints()`, which works similarly as the function `sum()`.

```
>>> prob.remove_all_constraints()       #we first remove the 4 constraints precedently added
>>> prob.add_constraint(Y>0)            #a single constraint
>>> prob.add_list_of_constraints([Z[i]==Z[i-1]+Y.T for i in range(1,5)],'i','1...4')
...                                     #the same list of constraints as above
>>> print prob
---------------------
optimization problem (MIP):
59 variables, 40 affine constraints

w   : dict of 6 variables, (1, 1), binary
Z   : list of 5 variables, (4, 2), continuous
t   : (1, 1), continuous
Y   : (2, 4), continuous
x   : (4, 1), integer

    maximize ⟨ A[0] | Y ⟩ -t
such that
  Y > |0|
  Z[i] = Z[i-1] + Y.T for all i in 1...4
---------------------
```

Now, the constraint `Z[3] = Z[2] + Y.T`, which has been entered in 4th position, can either be accessed by `prob.get_constraint(3)` (`3` because constraints are numbered from `0`), or by

```
>>> prob.get_constraint((1,2))
# (4x2)-affine constraint: Z[3] = Z[2] + Y.T #
```

where `(1,2)` means *the 3rd constraint of the 2d group of constraints*, with zero-based numbering.

Similarly, the constraint `Y > |0|` can be accessed by `prob.get_constraint(0)` (first constraint), `prob.get_constraint((0,0))` (first constraint of the first group), or `prob.get_constraint((0,))` (unique constraint of the first group).

2.5.3 Quadratic constraints

Quadratic inequalities are entered in the following way:

```
>>> t**2 > 2*t - alpha + x[1]*x[2]
#Quadratic constraint -t**2 + 2.0*t -alpha + x[1]*x[2] < 0 #
>>> (t & alpha) * A[1] * x + (x +2 | Z[1][:,1]) < 3*(1|Y)-alpha
#Quadratic constraint [t,alpha]*A[1]*x + ⟨ x + |2.0| | Z[1][:,1] ⟩ -(3.0*⟨ |1| | Y ⟩ -alpha) < 0 #
```

Note that PICOS does not check the convexity of convex constraints. It is the solver which will raise an Exception if it does not support non-convex quadratics.

2.5.4 Second Order Cone Constraints

There are two types of second order cone constraints supported in PICOS.

- The constraints of the type $\|x\| \leq t$, where t is a scalar affine expression and x is a multidimensional affine expression (possibly a matrix, in which case the norm is Frobenius). This inequality forces the vector $[x; t]$ to belong to a Lorrentz-Cone (also called *ice-cream cone*)
- The constraints of the type $\|x\|^2 \leq tu,\ t \geq 0$, where t and u are scalar affine expressions and x is a multidimensional affine expression, which constrain the vector $[x, t, u]$ inside a rotated version of the Lorretz cone. When a constraint of the form `abs(x)**2 < t*u` is passed to PICOS, **it is implicitely assumed that `t` is nonnegative**, and the constraint is handled as the equivalent, standard ice-cream cone constraint $\| [2x, t-u] \| \leq t+u$.

A few examples:

```
>>> abs(x) < (2|x-1)                                    #A simple ice-cream cone constraint
# (4x1)-SOC constraint: ||x|| < ⟨ |2.0| | x -|1| ⟩ #
>>> abs(Y+Z[0].T) < t+alpha                             #SOC constraint with Frobenius norm
# (2x4)-SOC constraint: ||Y + Z[0].T|| < t + alpha #
>>> abs(Z[1][:,0])**2 < (2*t-alpha)*(x[2]-x[-1])        #Rotated SOC constraint
# (4x1)-Rotated SOC constraint: ||Z[1][:,0]||^2 < ( 2.0*t -alpha)( x[2] -(x[-1])) #
>>> t**2 < D['Elisa']+t                                 #t**2 is understood as
...                                                     #the squared norm of [t]
# (1x1)-Rotated SOC constraint: ||t||^2 < D[Elisa] + t #
>>> 1 < (t-1)*(x[2]+x[3])                               #1 is understood as
...                                                     #the squared norm of [1]
# (1x1)-Rotated SOC constraint: 1.0 < ( t -1.0)( x[2] + x[3]) #
```

2.5.5 Semidefinite Constraints

Linear matrix inequalities (LMI) can be entered thanks to an overload of the operators << and >>. For example, the LMI

$$\sum_{i=0}^{3} x_i b_i b_i^T \succeq b_4 b_4^T,$$

where $\succeq$ is used to denote the Löwner ordering, is passed to PICOS by writing:

```
>>> pic.sum([x[i]*b[i]*b[i].T for i in range(4)],'i','0...3') >> b[4]*b[4].T
# (4x4)-LMI constraint Σ_{i in 0...3} x[i]*b[i]*b[i].T ≽ b[4]*b[4].T #
```

Note the difference with

```
>>> pic.sum([x[i]*b[i]*b[i].T for i in range(4)],'i','0...3') > b[4]*b[4].T
# (4x4)-affine constraint: Σ_{i in 0...3} x[i]*b[i]*b[i].T > b[4]*b[4].T #
```

which yields an elementwise inequality.

For convenience, it is possible to add a symmetric matrix variable `X`, by specifying the option `vtype=symmetric`. This has the effect to store all the affine expressions which depend on `X` as a function of its lower triangular elements only.

```
>>> sdp = pic.Problem()
>>> X = sdp.add_variable('X',(4,4),vtype='symmetric')
>>> sdp.add_constraint(X >> 0)
>>> print sdp
---------------------
optimization problem (SDP):
10 variables, 0 affine constraints, 10 vars in 1 SD cones

X   : (4, 4), symmetric

    find vars
such that
  X ≽ |0|
---------------------
```

In this example, you see indeed that the problem has 10=(4*5)/2 variables, which correspond to the lower triangular elements of `X`.

Warning: When a constraint of the form `A >> B` is passed to PICOS, it is not assumed that A-B is symmetric. Instead, the symmetric matrix whose lower triangular elements are those of A-B is forced to be positive semidefnite. So, in the cases where A-B is not implicitely forced to be symmetric, you should add a constraint of the form `A-B==(A-B).T` in the problem.

2.6 Write a Problem to a file

It is possible to write a problem to a file, thanks to the function `write_to_file()`. Several file formats and file writers are available, have a look at the doc of `write_to_file()` for more explanations.

Below is a *hello world* example, which writes a simple MIP to a **.lp** file:

```
import picos as pic
prob = pic.Problem()
y = prob.add_variable('y',1, vtype='integer')
x = prob.add_variable('x',1)
prob.add_constraint(x>1.5)
prob.add_constraint(y-x>0.7)
prob.set_objective('min',y)
#let first picos display the problem
print prob
print
#now write the problem to a .lp file...
prob.write_to_file('helloworld.lp')
print
#and display the content of the freshly created file:
print open('helloworld.lp').read()
```

Generated output:

```
---------------------
optimization problem  (MIP):
2 variables, 2 affine constraints

y   : (1, 1), integer
x   : (1, 1), continuous

        minimize y
such that
```

```
x > 1.5
y -x > 0.7
---------------------

writing problem in helloworld.lp...
done.

\* file helloworld.lp generated by picos*\
Minimize
obj : 1 y
Subject To
in0 : -1 y+ 1 x <= -0.7
Bounds
y free
1.5 <= x<= +inf
Generals
y
Binaries
End
```

2.7 Solve a Problem

To solve a problem, you have to use the method `solve()` of the class `Problem`. This method accepts several options. In particular the solver can be specified by passing an option of the form `solver='solver_name'`. For a list of available parameters with their default values, see the doc of the function `set_all_options_to_default()`.

Once a problem has been solved, the optimal values of the variables are accessible with the `value` property. Depending on the solver, you can also obtain the slack and the optimal dual variables of the constraints thanks to the properties `dual` and `slack` of the class `Constraint`. See the doc of `dual` for more explanations on the dual variables for second order cone programs (SOCP) and semidefinite programs (SDP).

The class `Problem` also has two interesting properties: `type`, which indicates the class of the optimization problem ('LP', 'SOCP', 'MIP', 'SDP',...), and `status`, which indicates if the problem has been solved (the default is `'unsolved'`; after a call to `solve()` this property can take the value of any code returned by a solver, such as `'optimal'`, `'unbounded'`, `'near-optimal'`, `'primal infeasible'`, `'unknown'`, ...).

Below is a simple example, to solve the linear programm:

$$\begin{array}{ll} \underset{x\in\mathbb{R}^2}{\text{minimize}} & 0.5x_1 + x_2 \\ \text{subject to} & \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} x \leq \begin{bmatrix} 3 \\ 4 \end{bmatrix}. \end{array}$$

More examples can be found *here*.

```
P = pic.Problem()
A = pic.new_param('A', cvx.matrix([[1,1],[0,1]]) )
x = P.add_variable('x',2)
P.add_constraint(x[0]>x[1])
P.add_constraint(A*x<[3,4])
objective = 0.5 * x[0] + x[1]
P.set_objective('max', objective)

#display the problem and solve it
print P
print 'type:   '+P.type
print 'status: '+P.status
P.solve(solver='cvxopt',verbose=False)
print 'status: '+P.status
```

```
#--------------------#
#  objective value   #
#--------------------#

print 'the optimal value of this problem is:'
print P.obj_value()                 #"print objective" would also work

#--------------------#
#  optimal variable  #
#--------------------#
x_opt = x.value
print 'The solution of the problem is:'
print x_opt                         #"print x" would also work, since x is now valued
print

#--------------------#
#  slacks and duals  #
#--------------------#
c0=P.get_constraint(0)
print 'The dual of the constraint'
print c0
print 'is:'
print c0.dual
print 'And its slack is:'
print c0.slack
print

c1=P.get_constraint(1)
print 'The dual of the constraint'
print c1
print 'is:'
print c1.dual
print 'And its slack is:'
print c1.slack
```

```
----------------------
optimization problem  (LP):
2 variables, 3 affine constraints

x   : (2, 1), continuous

    maximize 0.5*x[0] + x[1]
such that
  x[0] > x[1]
  A*x < [ 2 x 1 MAT ]
----------------------
type:   LP
status: unsolved
status: optimal

the optimal value of this problem is:
3.0000000002
The solution of the problem is:
[ 2.00e+00]
[ 2.00e+00]


The dual of the constraint
# (1x1)-affine constraint : x[0] > x[1] #
is:
[ 2.50e-01]
```

```
And its slack is:
[ 1.83e-09]


The dual of the constraint
# (2x1)-affine constraint : A*x < [ 2 x 1 MAT ] #
is:
[ 4.56e-10]
[ 7.50e-01]

And its slack is:
[ 1.00e+00]
[-8.71e-10]
```

2.7.1 A note on dual variables

For second order cone constraints of the form $\|\mathbf{x}\| \leq t$, where $\mathbf{x}$ is a vector of dimension n, the dual variable is a vector of dimension $n+1$ of the form $[\lambda; \mathbf{z}]$, where the $n-$ dimensional vector $\mathbf{z}$ satisfies $\|\mathbf{z}\| \leq \lambda$.

Since *rotated* second order cone constraints of the form $\|\mathbf{x}\|^2 \leq tu,\ t \geq 0$, are handled as the equivalent ice-cream constraint $\|[2\mathbf{x}; t-u]\| \leq t+u$, the dual is given with respect to this reformulated, standard SOC constraint.

In general, a linear problem with second order cone constraints (both standard and rotated) and semidefinite constraints can be written under the form:

$$
\begin{array}{lll}
\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^T\mathbf{x} & \\
\text{subject to} & A^e\mathbf{x} + \mathbf{b^e} = 0 & \\
& A^l\mathbf{x} + \mathbf{b^l} \leq 0 & \\
& \|A_i^s\mathbf{x} + \mathbf{b_i^s}\| \leq \mathbf{f_i^s}^T\mathbf{x} + d_i^s, & \forall i \in I \\
& \|A_j^r\mathbf{x} + \mathbf{b_j^r}\|^2 \leq (\mathbf{f_j^{r_1}}^T\mathbf{x} + d_j^{r_1})(\mathbf{f_j^{r_2}}^T\mathbf{x} + d_j^{r_2}), & \forall j \in J \\
& 0 \leq \mathbf{f_j^{r_1}}^T\mathbf{x} + d_j^{r_1}, & \forall j \in J \\
& \sum_{i=1}^n x_i M_i \succeq M_0 &
\end{array}
$$

where

- $\mathbf{c}, \left\{\mathbf{f_i^s}\right\}_{i\in I}, \left\{\mathbf{f_j^{r_1}}\right\}_{j\in J}, \left\{\mathbf{f_j^{r_2}}\right\}_{j\in J}$ are vectors of dimension n;
- $\left\{d_i^s\right\}_{i\in I}, \left\{d_j^{r_1}\right\}_{j\in J}, \left\{d_j^{r_2}\right\}_{j\in J}$ are scalars;
- $\left\{\mathbf{b_i^s}\right\}_{i\in I}$ are vectors of dimension n_i^s and $\left\{A_i^s\right\}_{i\in I}$ are matrices of size $n_i^s \times n$;
- $\left\{\mathbf{b_j^r}\right\}_{j\in J}$ are vectors of dimension n_j^r and $\left\{A_j^r\right\}_{j\in J}$ are matrices of size $n_j^r \times n$;
- $\mathbf{b^e}$ is a vector of dimension n^e and A^e is a matrix of size $n^e \times n$;
- $\mathbf{b^l}$ is a vector of dimension n^l and A^l is a matrix of size $n^l \times n$;
- $\left\{M_k\right\}_{k=0,\ldots,n}$ are $m \times m$ symmetric matrices ($M_k \in$: $math$:mathbb{S}_m‘).

Its dual problem can be written as:

$$
\begin{array}{lll}
\text{maximize} & \mathbf{b^e}^T\mu^{\mathbf{e}} + \mathbf{b^l}^T\mu^{\mathbf{l}} & + \sum_{i\in I}\left(\mathbf{b_i^s}^T\mathbf{z_i^s} - d_i^s\lambda_i\right) + \sum_{j\in J}\left(\mathbf{b_j^r}^T\mathbf{z_j^r} - d_j^{r_1}\alpha_j - d_j^{r_2}\beta_j\right) + \langle M_0, X\rangle \\
\text{subject to} & c + A^{e^T}\mu^{\mathbf{e}} + A^{l^T}\mu^{\mathbf{l}} & + \sum_{i\in I}\left(A_i^{s^T}\mathbf{z_i^s} - \lambda_i\mathbf{f_i^s}\right) + \sum_{j\in J}\left(A_j^{r^T}\mathbf{z_j^r} - \alpha_j\mathbf{f_j^{r_1}} - \beta_j\mathbf{f_j^{r_2}}\right) = \mathcal{M} \bullet X \\
& \mu_l \geq 0 & \\
& \|\mathbf{z_i^s}\| \leq \lambda_i, & \forall i \in I \\
& \|\mathbf{z_j^r}\|^2 \leq 4\alpha_j\beta_j, & \forall j \in J \\
& 0 \leq \alpha_j, & \forall j \in J \\
& X \succeq 0 &
\end{array}
$$

where $\mathcal{M} \bullet X$ stands for the vector of dimension n with $\langle M_i, X\rangle$ on the i th coordinate, and the dual variables are

- $\mu^e \in \mathbb{R}^{n_e}$
- $\mu^l \in \mathbb{R}^{n_l}$

- $z_i^s \in \mathbb{R}^{n_i^s},\ \forall i \in I$
- $\lambda_i \in \mathbb{R},\ \forall i \in I$
- $z_j^r \in \mathbb{R}^{n_j^r},\ \forall j \in J$
- $(\alpha_j, \beta_j) \in \mathbb{R}^2,\ \forall j \in J$
- $X \in \mathbb{S}_m$

When quering the dual of a constraint of the above primal problem, **picos will return**

- μ^e for the constraint $A^e\mathbf{x} + \mathbf{b^e} = 0$;
- μ^l for the constraint $A^l\mathbf{x} + \mathbf{b^l} \geq 0$;
- The $(n_i^s + 1)-$ dimensional vector $\mu_i^s := [\lambda_i; \mathbf{z_i^s}]$ for the constraint

  $$\|A_i^s\mathbf{x} + \mathbf{b_i^s}\| \leq \mathbf{f_i^s}^T\mathbf{x} + d_i^s;$$

- The $(n_j^r + 2)-$ dimensional vector $\mu_j^r = \frac{1}{2}[\ (\beta_j + \alpha_j);\ \mathbf{z_j^r};\ (\beta_j - \alpha_j)\ ]$ for the constraint

  $$\|A_j^r\mathbf{x} + \mathbf{b_j^r}\|^2 \leq (\mathbf{f_j^{r_1}}^T\mathbf{x} + d_j^{r_1})(\mathbf{f_j^{r_2}}^T\mathbf{x} + d_j^{r_2})$$

  In other words, if the dual vector returned by picos is of the form $\mu_j^r = [\sigma_j^1; \mathbf{u_j}; \sigma_j^2]$, where $\mathbf{u_j}$ is of dimension n_j^r, then the dual variables of the rotated conic constraint are $\alpha_j = \sigma_j^1 - \sigma_j^2,\ \beta_j = \sigma_j^1 + \sigma_j^2$ and $\mathbf{z_j^r} = 2\mathbf{u_j}$;

- The symmetric positive definite matrix X for the constraint

  $$\sum_{i=1}^n x_i M_i \succeq M_0.$$

CHAPTER

THREE

EXAMPLES

3.1 Examples from Optimal Experimental Design

Optimal experimental design is a theory at the interface of statistics and optimization, which studies how to allocate some experimental effort within a set of available expeiments. The goal is to allow for the best possible estimation of an unknown parameter θ. In what follows, we assume the standard linear model with multiresponse experiments: the i^{th} experiment gives a multidimensional observation that can be written as $y_i = A_i^T\theta + \epsilon_i$, where y_i is of dimension l_i, A_i is a $m \times l_i-$ matrix, and the noise vectors ϵ_i are i.i.d. with a unit variance.

Several optimization criterions exist, leading to different SDP, SOCP and LP formulations. As such, optimal experimental design problens are natural examples for problems in conic optimization. For a review of the different formulations and more references, see *[1]*.

The code below initializes the data used in all the examples of this page. It should be run prior to any of the codes presented in this page.

```
import cvxopt as cvx
import picos as pic

#---------------------------------#
# First generate some data :      #
#       _ a list of 8 matrices A  #
#       _ a vector c              #
#---------------------------------#
A=[ cvx.matrix([[1,0,0,0,0],
                [0,3,0,0,0],
                [0,0,1,0,0]]),
cvx.matrix([[0,0,2,0,0],
                [0,1,0,0,0],
                [0,0,0,1,0]]),
cvx.matrix([[0,0,0,2,0],
                [4,0,0,0,0],
                [0,0,1,0,0]]),
cvx.matrix([[1,0,0,0,0],
                [0,0,2,0,0],
                [0,0,0,0,4]]),
cvx.matrix([[1,0,2,0,0],
                [0,3,0,1,2],
                [0,0,1,2,0]]),
cvx.matrix([[0,1,1,1,0],
                [0,3,0,1,0],
                [0,0,2,2,0]]),
cvx.matrix([[1,2,0,0,0],
                [0,3,3,0,5],
                [1,0,0,2,0]]),
cvx.matrix([[1,0,3,0,1],
                [0,3,2,0,0],
                [1,0,0,2,0]])
```

```
]

c = cvx.matrix([1,2,3,4,5])
```

3.1.1 c-optimality, multi-response: SOCP

We compute the c-optimal design (`c=[1,2,3,4,5]`) for the observation matrices `A[i].T` from the variable `A` defined above. The results below suggest that we should allocate 12.8% of the experimental effort on experiment #5, and 87.2% on experiment #7.

Primal Problem

The SOCP for multiresponse c-optimal design is:

$$\begin{aligned} \underset{\substack{\mu\in\mathbb{R}^s \\ \forall i\in[s],\ z_i\in\mathbb{R}^{l_i}}}{\text{minimize}} \quad & \sum_{i=1}^{s} \mu_i \\ \text{subject to} \quad & \sum_{i=1}^{s} A_i z_i = c \\ & \forall i \in [s],\ \|z_i\|_2 \leq \mu_i, \end{aligned}$$

```
#create the problem, variables and params
prob_primal_c=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
z=[prob_primal_c.add_variable('z['+str(i)+']',AA[i].size[1]) for i in range(s)]
mu=prob_primal_c.add_variable('mu',s)

#define the constraints and objective function
prob_primal_c.add_list_of_constraints(
        [abs(z[i])<mu[i] for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_primal_c.add_constraint(
        pic.sum(
                [AA[i]*z[i] for i in range(s)], #summands
                'i', #index
                '[s]' #set to which the index belongs
                )
        == cc )
prob_primal_c.set_objective('min',1|mu)

#solve the problem and retrieve the optimal weights of the optimal design.
print prob_primal_c
prob_primal_c.solve(verbose=0,solver='cvxopt')

mu=mu.value
w=mu/sum(mu) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
32 variables, 5 affine constraints, 32 vars in 8 SO cones

z   : list of 8 variables, (3, 1), continuous
mu  : (8, 1), continuous

        minimize ⟨ |1| | mu ⟩
such that
||z[i]|| < mu[i] for all i in [s]
Σ_{i in [s]} A[i]⋆z[i] = c
---------------------

The optimal deign is:
[...]
[...]
[...]
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]
```

The `[...]` above indicate a numerical zero entry (*i.e., which can be something like 2.84e-10*). We use the ellipsis `...` instead for clarity and compatibility with **doctest**.

Dual Problem

This is only to check that we obtain the same solution with the dual problem, and to provide one additional example in this doc:

$$\begin{aligned} \underset{u \in \mathbb{R}^m}{\text{maximize}} \quad & c^T u \\ \text{subject to} \quad & \forall i \in [s],\ \|A_i^T u\|_2 \leq 1 \end{aligned}$$

```
#create the problem, variables and params
prob_dual_c=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
u=prob_dual_c.add_variable('u',c.size)

#define the constraints and objective function
prob_dual_c.add_list_of_constraints(
        [abs(AA[i].T⋆u)<1 for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_dual_c.set_objective('max', cc|u)

#solve the problem and retrieve the weights of the optimal design
print prob_dual_c
prob_dual_c.solve(verbose=0)

#Lagrangian duals of the SOC constraints
mu = [cons.dual[0] for cons in prob_dual_c.get_constraint((0,))]
mu = cvx.matrix(mu)
w=mu/sum(mu) #normalize mu to get the optimal weights
print
```

```
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
5 variables, 0 affine constraints, 32 vars in 8 SO cones

u   : (5, 1), continuous

        maximize ⟨ c | u ⟩
such that
||A[i].T*u|| < 1 for all i in [s]
---------------------

The optimal deign is:
[...]
[...]
[...]
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]
```

3.1.2 c-optimality, single-response: LP

When the observation matrices are row vectors (single-response framework), the SOCP above reduces to a simple LP, because the variables z_i are scalar. We solve below the LP for the case where there are 12 available experiments, corresponding to the columns of the matrices `A[4]`, `A[5]`, `A[6]`, and `A[7]` defined in the preambule.

The optimal design allocates 3.37% to experiment #5 (2nd column of `A[5]`), 27.9% to experiment #7 (1st column of `A[6]`), 11.8% to experiment #8 (2nd column of `A[6]`), 27.6% to experiment #9 (3rd column of `A[6]`), and 29.3% to experiment #11 (2nd column of `A[7]`).

```
#create the problem, variables and params
prob_LP=pic.Problem()
AA=[cvx.sparse(a[:,i],tc='d') for i in range(3) for a in A[4:]] #12 column vectors
s=len(AA)
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
z=[prob_LP.add_variable('z['+str(i)+']',1) for i in range(s)]
mu=prob_LP.add_variable('mu',s)

#define the constraints and objective function
prob_LP.add_list_of_constraints(
        [abs(z[i])<mu[i] for i in range(s)], #constraints handled as -mu_i < z_i< mu_i
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_LP.add_constraint(
        pic.sum(
                [AA[i]*z[i] for i in range(s)], #summands
                'i', #index
                '[s]' #set to which the index belongs
                )
        == cc )
prob_LP.set_objective('min',1|mu)

#solve the problem and retrieve the weights of the optimal design
print prob_LP
```

```
prob_LP.solve(verbose=0)

mu=mu.value
w=mu/sum(mu) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Note that there are no cone constraints, because the constraints of the form $|z_i| \leq \mu_i$ are handled as two inequalities when z_i is scalar, so the problem is a LP indeed:

```
---------------------
optimization problem  (LP):
24 variables, 29 affine constraints

z   : list of 12 variables, (1, 1), continuous
mu  : (12, 1), continuous

        minimize ⟨ |1| | mu ⟩
such that
||z[i]|| < mu[i] for all i in [s]
Σ_{i in [s]} A[i]⋆z[i] = c
---------------------

The optimal deign is:
[...]
[...]
[...]
[...]
[ 3.37e-02]
[...]
[ 2.79e-01]
[ 1.18e-01]
[ 2.76e-01]
[...]
[ 2.93e-01]
[...]
```

3.1.3 SDP formulation of the c-optimal design problem

We give below the SDP for c-optimality, in primal and dual form. You can observe that we obtain the same results as with the SOCP presented earlier: 12.8% on experiment #5, and 87.2% on experiment #7.

Primal Problem

The SDP formulation of the c-optimal design problem is:

$$\begin{aligned} \underset{\mu \in \mathbb{R}^s}{\text{minimize}} \quad & \sum_{i=1}^{s} \mu_i \\ \text{subject to} \quad & \sum_{i=1}^{s} \mu_i A_i A_i^T \succeq cc^T, \\ & \mu \geq 0. \end{aligned}$$

```
#create the problem, variables and params
prob_SDP_c_primal=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
```

```
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
mu=prob_SDP_c_primal.add_variable('mu',s)

#define the constraints and objective function
prob_SDP_c_primal.add_constraint(
        pic.sum(
        [mu[i]*AA[i]*AA[i].T for i in range(s)], #summands
        'i', #index
        '[s]' #set to which the index belongs
        )
        >> cc*cc.T )
prob_SDP_c_primal.add_constraint(mu>0)
prob_SDP_c_primal.set_objective('min',1|mu)

#solve the problem and retrieve the weights of the optimal design
print prob_SDP_c_primal
prob_SDP_c_primal.solve(verbose=0)
w=mu.value
w=w/sum(w) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SDP):
8 variables, 8 affine constraints, 15 vars in 1 SD cones

mu  : (8, 1), continuous

        minimize ⟨ |1| | mu ⟩
such that
Σ_{i in [s]} mu[i]*A[i]*A[i].T ≽ c*c.T
mu > |0|
---------------------

The optimal deign is:
[...]
[...]
[...]
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]
```

Dual Problem

This is only to check that we obtain the same solution with the dual problem, and to provide one additional example in this doc:

$$\begin{array}{ll} \underset{X \in \mathbb{R}^{m \times m}}{\text{maximize}} & c^T X c \\ \text{subject to} & \forall i \in [s],\ \langle A_i A_i^T,\ X \rangle \leq 1, \\ & X \succeq 0. \end{array}$$

```
#create the problem, variables and params
prob_SDP_c_dual=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
m =c.size[0]
X=prob_SDP_c_dual.add_variable('X',(m,m),vtype='symmetric')

#define the constraints and objective function
prob_SDP_c_dual.add_list_of_constraints(
        [(AA[i]*AA[i].T | X ) <1 for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_SDP_c_dual.add_constraint(X>>0)
prob_SDP_c_dual.set_objective('max', cc.T*X*cc)

#solve the problem and retrieve the weights of the optimal design
print prob_SDP_c_dual
prob_SDP_c_dual.solve(verbose=0,solver='smcp')
#Lagrangian duals of the SOC constraints
mu = [cons.dual[0] for cons in prob_SDP_c_dual.get_constraint((0,))]
mu = cvx.matrix(mu)
w=mu/sum(mu) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
print 'and the optimal positive semidefinite matrix X is'
print X
```

Generated output:

```
---------------------
optimization problem  (SDP):
15 variables, 8 affine constraints, 15 vars in 1 SD cones

X   : (5, 5), symmetric

        maximize c.T*X*c
such that
⟨ A[i]*A[i].T | X ⟩ < 1.0 for all i in [s]
X ≽ |0|
---------------------

The optimal deign is:
[...]
[...]
[...]
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]

and the optimal positive semidefinite matrix X is
[ 5.92e-03  8.98e-03  2.82e-03 -3.48e-02 -1.43e-02]
[ 8.98e-03  1.36e-02  4.27e-03 -5.28e-02 -2.17e-02]
[ 2.82e-03  4.27e-03  1.34e-03 -1.66e-02 -6.79e-03]
[-3.48e-02 -5.28e-02 -1.66e-02  2.05e-01  8.39e-02]
[-1.43e-02 -2.17e-02 -6.79e-03  8.39e-02  3.44e-02]
```

3.1.4 A-optimality: SOCP

We compute the A-optimal design for the observation matrices `A[i].T` defined in the preambule. The optimal design allocates 24.9% on experiment #3, 14.2% on experiment #4, 8.51% on experiment #5, 12.1% on experiment #6, 13.2% on experiment #7, and 27.0% on experiment #8.

[2.49e-01] [1.42e-01] [8.51e-02] [1.21e-01] [1.32e-01] [2.70e-01]

Primal Problem

The SOCP for the A-optimal design problem is:

$$\begin{aligned} \underset{\substack{\mu\in\mathbb{R}^s \\ \forall i\in[s],\ Z_i\in\mathbb{R}^{l_i\times m}}}{\text{minimize}} \quad & \sum_{i=1}^{s} \mu_i \\ \text{subject to} \quad & \sum_{i=1}^{s} A_i Z_i = I \\ & \forall i \in [s],\ \|Z_i\|_F \leq \mu_i, \end{aligned}$$

```
#create the problem, variables and params
prob_primal_A=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
AA=pic.new_param('A',AA)
Z=[prob_primal_A.add_variable('Z['+str(i)+']',AA[i].T.size) for i in range(s)]
mu=prob_primal_A.add_variable('mu',s)

#define the constraints and objective function
prob_primal_A.add_list_of_constraints(
        [abs(Z[i])<mu[i] for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_primal_A.add_constraint(
        pic.sum(
        [AA[i]*Z[i] for i in range(s)], #summands
        'i', #index
        '[s]' #set to which the index belongs
        )
        == 'I' )
prob_primal_A.set_objective('min',1|mu)

#solve the problem and retrieve the weights of the optimal design
print prob_primal_A
prob_primal_A.solve(verbose=0)
w=mu.value
w=w/sum(w) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
128 variables, 25 affine constraints, 128 vars in 8 SO cones

Z   : list of 8 variables, (3, 5), continuous
mu  : (8, 1), continuous
```

```
        minimize ⟨ |1| | mu ⟩
such that
||Z[i]|| < mu[i] for all i in [s]
Σ_{i in [s]} A[i]⋆Z[i] = I
---------------------

The optimal deign is:
[...]
[...]
[ 2.49e-01]
[ 1.42e-01]
[ 8.51e-02]
[ 1.21e-01]
[ 1.32e-01]
[ 2.70e-01]
```

Dual Problem

This is only to check that we obtain the same solution with the dual problem, and to provide one additional example in this doc:

$$\begin{array}{ll} \underset{U \in \mathbb{R}^{m \times m}}{\text{maximize}} & \text{trace } U \\ \text{subject to} & \forall i \in [s],\ \|A_i^T U\|_2 \leq 1 \end{array}$$

```
#create the problem, variables and params
prob_dual_A=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
m=AA[0].size[0]
AA=pic.new_param('A',AA)
U=prob_dual_A.add_variable('U',(m,m))

#define the constraints and objective function
prob_dual_A.add_list_of_constraints(
        [abs(AA[i].T⋆U)<1 for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_dual_A.set_objective('max', 'I'|U)

#solve the problem and retrieve the weights of the optimal design
print prob_dual_A
prob_dual_A.solve(verbose = 0)

#Lagrangian duals of the SOC constraints
mu = [cons.dual[0] for cons in prob_dual_A.get_constraint((0,))]
mu = cvx.matrix(mu)
w=mu/sum(mu) #normalize mu to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
25 variables, 0 affine constraints, 128 vars in 8 SO cones
```

```
U   : (5, 5), continuous

        maximize trace( U )
such that
||A[i].T*U|| < 1 for all i in [s]
---------------------

The optimal deign is:
[...]
[...]
[ 2.49e-01]
[ 1.42e-01]
[ 8.51e-02]
[ 1.21e-01]
[ 1.32e-01]
[ 2.70e-01]
```

3.1.5 A-optimality with multiple constraints: SOCP

A-optimal designs can also be computed by SOCP when the vector of weights $\mathbf{w}$ is subject to several linear constraints. To give an example, we compute the A-optimal design for the observation matrices given in the preambule, when the weights must satisfy: $\sum_{i=0}^{3} w_i \leq 0.5$ and $\sum_{i=4}^{7} w_i \leq 0.5$. This problem has the following SOCP formulation:

$$
\begin{array}{ll}
\underset{\substack{\mathbf{w}\in\mathbb{R}^s \\ \mu\in\mathbb{R}^s \\ \forall i\in[s],\ Z_i\in\mathbb{R}^{l_i\times m}}}{\text{minimize}} & \displaystyle\sum_{i=1}^{s} \mu_i \\
\text{subject to} & \displaystyle\sum_{i=1}^{s} A_i Z_i = I \\
& \displaystyle\sum_{i=0}^{3} w_i \leq 0.5 \\
& \displaystyle\sum_{i=4}^{7} w_i \leq 0.5 \\
& \forall i \in [s],\ \|Z_i\|_F^2 \leq \mu_i w_i,
\end{array}
$$

The optimal solution allocates 29.7% and 20.3% to the experiments #3 and #4, and respectively 6.54%, 11.9%, 9.02% and 22.5% to the experiments #5 to #8:

```
#create the problem, variables and params
prob_A_multiconstraints=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
AA=pic.new_param('A',AA)

mu=prob_A_multiconstraints.add_variable('mu',s)
w =prob_A_multiconstraints.add_variable('w',s)
Z=[prob_A_multiconstraints.add_variable('Z['+str(i)+']',AA[i].T.size) for i in range(s)]

#define the constraints and objective function
prob_A_multiconstraints.add_constraint(
        pic.sum(
        [AA[i]*Z[i] for i in range(s)], #summands
        'i', #index
        '[s]' #set to which the index belongs
        )
```

```
        == 'I' )
prob_A_multiconstraints.add_constraint( (1|w[:4]) < 0.5)
prob_A_multiconstraints.add_constraint( (1|w[4:]) < 0.5)
prob_A_multiconstraints.add_list_of_constraints(
                [abs(Z[i])**2<mu[i]*w[i]
                for i in range(s)],'i','[s]')
prob_A_multiconstraints.set_objective('min',1|mu)

#solve the problem and retrieve the weights of the optimal design
print prob_A_multiconstraints
prob_A_multiconstraints.solve(verbose=0)
w=w.value
w=w/sum(w) #normalize w to get the optimal weights
print
print 'The optimal deign is:'
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
136 variables, 27 affine constraints, 136 vars in 8 SO cones

Z   : list of 8 variables, (3, 5), continuous
mu  : (8, 1), continuous
w   : (8, 1), continuous

        minimize ⟨ |1| | mu ⟩
such that
Σ_{i in [s]} A[i]*Z[i] = I
⟨ |1| | w[:4] ⟩ < 0.5
⟨ |1| | w[4:] ⟩ < 0.5
||Z[i]||^2 < ( mu[i])( w[i]) for all i in [s]
---------------------

The optimal deign is:
[...]
[...]
[ 2.97e-01]
[ 2.03e-01]
[ 6.54e-02]
[ 1.19e-01]
[ 9.02e-02]
[ 2.25e-01]
```

3.1.6 Exact A-optimal design: MISOCP

In the exact version of A-optimality, a number $N \in \mathbb{N}$ of experiments is given, and the goal is to find the optimal number of times $n_i \in \mathbb{N}$ that the experiment #i should be performed, with $\sum_i n_i = N$.

The SOCP formulation of A-optimality for constrained designs also accept integer constraints, which results in a MISOCP for exact A-optimality:

$$\begin{aligned}
\underset{\substack{\mathbf{t}\in\mathbb{R}^s\\ \mathbf{n}\in\mathbb{N}^s\\ \forall i\in[s],\ Z_i\in\mathbb{R}^{l_i\times m}}}{\text{minimize}} \quad & \sum_{i=1}^{s} t_i \\
\text{subject to} \quad & \sum_{i=1}^{s} A_i Z_i = I \\
& \forall i \in [s],\ \|Z_i\|_F^2 \leq n_i t_i, \\
& \sum_{i=1}^{s} n_i = N.
\end{aligned}$$

The exact optimal design is $\mathbf{n} = [0, 0, 5, 3, 2, 2, 3, 5]$:

```
#create the problem, variables and params
prob_exact_A=pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
m=AA[0].size[0]
AA=pic.new_param('A',AA)
cc=pic.new_param('c',c)
N =pic.new_param('N',20) #number of experiments allowed
I =pic.new_param('I',cvx.spmatrix([1]*m,range(m),range(m),(m,m))) #identity matrix
Z=[prob_exact_A.add_variable('Z['+str(i)+']',AA[i].T.size) for i in range(s)]
n=prob_exact_A.add_variable('n',s, vtype='integer')
t=prob_exact_A.add_variable('t',s)

#define the constraints and objective function
prob_exact_A.add_list_of_constraints(
        [abs(Z[i])**2<n[i]*t[i] for i in range(s)], #constraints
        'i', #index
        '[s]' #set to which the index belongs
        )
prob_exact_A.add_constraint(
        pic.sum(
        [AA[i]*Z[i] for i in range(s)], #summands
        'i', #index
        '[s]' #set to which the index belongs
        )
        == I )

prob_exact_A.add_constraint( 1|n < N )
prob_exact_A.set_objective('min',1|t)

#solve the problem and display the optimal design
print prob_exact_A
prob_exact_A.solve(solver='mosek',verbose = 0)
print n
```

Generated output:

```
---------------------
optimization problem  (MISOCP):
136 variables, 26 affine constraints, 136 vars in 8 SO cones

Z       : list of 8 variables, (3, 5), continuous
n       : (8, 1), integer
t       : (8, 1), continuous

        minimize ⟨ |1| | t ⟩
such that
||Z[i]||^2 < ( n[i])( t[i]) for all i in [s]
```

```
Σ_{i in [s]} A[i]*Z[i] = I
⟨ |1| | n ⟩ < N
---------------------
[...]
[...]
[ 5.00e+00]
[ 3.00e+00]
[ 2.00e+00]
[ 2.00e+00]
[ 3.00e+00]
[ 5.00e+00]
```

3.1.7 approximate and exact D-optimal design: (MI)SOCP

The D-optimal design problem has a convex programming formulation:

$$
\begin{array}{ll}
\underset{\substack{\mathbf{L}\in\mathbb{R}^{m\times m}\\ \mathbf{w}\in\mathbb{R}^{s}\\ \forall i\in[s],\, V_i\in\mathbb{R}^{l_i\times m}}}{\text{maximize}} & \log\displaystyle\prod_{i=1}^{m} L_{i,i} \\
\text{subject to} & \displaystyle\sum_{i=1}^{s} A_i V_i = L, \\
& L \text{ lower triangular}, \\
& \|V_i\|_F \leq \sqrt{m}\, w_i, \\
& \displaystyle\sum_{i=1}^{s} w_i \leq 1.
\end{array}
$$

By introducing new SOC constraints, we can create a variable u_{01234} such that $u_{01234}^8 \leq \prod_{i=0}^{4} L_{i,i}$. Hence, the D-optimal problem can be solved by second order cone programming. The example below allocates respectively 22.7%, 3.38%, 1.65%, 5.44%, 31.8% and 35.1% to the experiments #3 to #8.

```
#create the problem, variables and params
prob_D = pic.Problem()
AA=[cvx.sparse(a,tc='d') for a in A] #each AA[i].T is a 3 x 5 observation matrix
s=len(AA)
m=AA[0].size[0]
AA=pic.new_param('A',AA)
mm=pic.new_param('m',m)
L=prob_D.add_variable('L',(m,m))
V=[prob_D.add_variable('V['+str(i)+']',AA[i].T.size) for i in range(s)]
w=prob_D.add_variable('w',s)
#additional variables to handle the product of the diagonal elements of L
u={}
for k in ['01','23','4.','0123','4...','01234']:
        u[k] = prob_D.add_variable('u['+k+']',1)


#define the constraints and objective function
prob_D.add_constraint(
                pic.sum([AA[i]*V[i]
                for i in range(s)],'i','[s]')
                == L)
#L is lower triangular
prob_D.add_list_of_constraints( [L[i,j] == 0
                                 for i in range(m)
                                 for j in range(i+1,m)],['i','j'],'upper triangle')
prob_D.add_list_of_constraints([abs(V[i])<(mm**0.5)*w[i]
```

```
                                for i in range(s)],'i','[s]')
prob_D.add_constraint(1|w<1)
#SOC constraints to define u['01234'] such that
#u['01234']**8 < L[0,0] * L[1,1] * ... * L[4,4]
prob_D.add_constraint(u['01']**2   <L[0,0]*L[1,1])
prob_D.add_constraint(u['23']**2   <L[2,2]*L[3,3])
prob_D.add_constraint(u['4.']**2   <L[4,4])
prob_D.add_constraint(u['0123']**2 <u['01']*u['23'])
prob_D.add_constraint(u['4...']**2 <u['4.'])
prob_D.add_constraint(u['01234']**2<u['0123']*u['4...'])

prob_D.set_objective('max',u['01234'])

#solve the problem and display the optimal design
print prob_D
prob_D.solve(verbose=0)
print w
```

Generated output:

```
---------------------
optimization problem  (SOCP):
159 variables, 36 affine constraints, 146 vars in 14 SO cones

V   : list of 8 variables, (3, 5), continuous
u   : dict of 6 variables, (1, 1), continuous
L   : (5, 5), continuous
w   : (8, 1), continuous

        maximize u[01234]
such that
L = Σ_{i in [s]} A[i]*V[i]
L[i,j] = 0 for all (i,j) in upper triangle
||V[i]|| < (m)**0.5*w[i] for all i in [s]
⟨ |1| | w ⟩ < 1.0
||u[01]||^2 < ( L[0,0])( L[1,1])
||u[23]||^2 < ( L[2,2])( L[3,3])
||u[4.]||^2 < L[4,4]
||u[0123]||^2 < ( u[01])( u[23])
||u[4...]||^2 < u[4.]
||u[01234]||^2 < ( u[0123])( u[4...])
---------------------
[...]
[...]
[ 2.27e-01]
[ 3.38e-02]
[ 1.65e-02]
[ 5.44e-02]
[ 3.18e-01]
[ 3.51e-01]
```

As for the A-optimal problem, there is an alternative SOCP formulation of D-optimality *[2]*, in which integer constraints may be added. This allows us to formulate the exact D-optimal problem as a MISOCP. For $N = 20$, we obtain the following N-exact D-optimal design: $\mathbf{n} = [0, 0, 5, 1, 0, 1, 6, 7]$:

```
#create the problem, variables and params
prob_exact_D = pic.Problem()
L=prob_exact_D.add_variable('L',(m,m))
V=[prob_exact_D.add_variable('V['+str(i)+']',AA[i].T.size) for i in range(s)]
T=prob_exact_D.add_variable('T',(s,m))
n=prob_exact_D.add_variable('n',s,'integer')
N = pic.new_param('N',20)
#additional variables to handle the product of the diagonal elements of L
u={}
```

```
for k in ['01','23','4.','0123','4...','01234']:
        u[k] = prob_exact_D.add_variable('u['+k+']',1)


#define the constraints and objective function
prob_exact_D.add_constraint(
                pic.sum([AA[i]*V[i]
                for i in range(s)],'i','[s]')
                == L)
#L is lower triangular
prob_exact_D.add_list_of_constraints( [L[i,j] == 0
                                for i in range(m)
                                for j in range(i+1,m)],['i','j'],'upper triangle')

prob_exact_D.add_list_of_constraints([abs(V[i][:,k])**2<n[i]/N*T[i,k]
                for i in range(s) for k in range(m)],['i','k'])

prob_exact_D.add_list_of_constraints([(1|T[:,k])<1
                for k in range(m)],'k')


prob_exact_D.add_constraint(1|n<N)

#SOC constraints to define u['01234'] such that
#u['01234']**8 < L[0,0] * L[1,1] * ... * L[4,4]
prob_exact_D.add_constraint(u['01']**2   <L[0,0]*L[1,1])
prob_exact_D.add_constraint(u['23']**2   <L[2,2]*L[3,3])
prob_exact_D.add_constraint(u['4.']**2   <L[4,4])
prob_exact_D.add_constraint(u['0123']**2 <u['01']*u['23'])
prob_exact_D.add_constraint(u['4...']**2 <u['4.'])
prob_exact_D.add_constraint(u['01234']**2<u['0123']*u['4...'])

prob_exact_D.set_objective('max',u['01234'])

#solve the problem and display the optimal design
print prob_exact_D
prob_exact_D.solve(solver='mosek',verbose=0)
print n
```

Generated output:

```
---------------------
optimization problem  (MISOCP):
199 variables, 41 affine constraints, 218 vars in 46 SO cones

V       : list of 8 variables, (3, 5), continuous
u       : dict of 6 variables, (1, 1), continuous
L       : (5, 5), continuous
T       : (8, 5), continuous
n       : (8, 1), integer

        maximize u[01234]
such that
L = Σ_{i in [s]} A[i]*V[i]
L[i,j] = 0 for all (i,j) in upper triangle
||V[i][:,k]||^2 < ( n[i] / N)( T[i,k]) for all (i,k)
⟨ |1| | T[:,k] ⟩ < 1.0 for all k
⟨ |1| | n ⟩ < N
||u[01]||^2 < ( L[0,0])( L[1,1])
||u[23]||^2 < ( L[2,2])( L[3,3])
||u[4.]||^2 < L[4,4]
||u[0123]||^2 < ( u[01])( u[23])
||u[4...]||^2 < u[4.]
```

```
||u[01234]||^2 < ( u[0123])( u[4...])
----------------------
[...]
[...]
[ 5.00e+00]
[ 1.00e+00]
[...]
[ 1.00e+00]
[ 6.00e+00]
[ 7.00e+00]
```

3.1.8 References

1. "Computing Optimal Designs of multiresponse Experiments reduces to Second-Order Cone Programming", G. Sagnol, *Journal of Statistical Planning and Inference*, 141(5), p. *1684-1708*, 2011.
2. "*SOC-representability of the D-criterion of optimal experimental design*", R. Harman and G. Sagnol, Draft.

3.2 Cut problems in graphs

The code below initializes the graph used in all the examples of this page. It should be run prior to any of the codes presented in this page. The packages networkx and matplotlib are recquired.

We use an arbitrary graph generated by the LCF generator of the networkx package. The graph is deterministic, so that we can run doctest and check the output. We also use a kind of arbitrary sequence for the edge capacities.

```
import picos as pic
import networkx as nx

#number of nodes
N=20

#Generate a graph with LCF notation
#(you can change the values below to obtain another graph!)
G=nx.LCF_graph(N,[1,3,14],5)
G=nx.DiGraph(G) #edges are bidirected

#generate edge capacities
c={}
for i,e in enumerate(G.edges()):
        c[e]=((-2)**i)%17 #an arbitrary sequence of numbers
```

3.2.1 Max-flow, Min-cut (LP)

Max-flow

Given a directed graph $G(V, E)$, with a capacity $c(e)$ on each edge $e \in E$, a source node s and a sink node t, the **max-flow** problem is to find a flow from s to t of maximum value. Recall that a flow s to t is a mapping from E to $\mathbb{R}^+$ such that:

- the capacity of each edge is respected: $\forall e \in E,\ f(e) \leq c(e)$
- the flow is conserved at each non-terminal node: $\forall n \in V \setminus \{s, t\},\ \sum_{(i,n)\in E} f((i, n)) = \sum_{(n,j)\in E} f((n, j))$

Its value is defined as the volume passing from s to t:

$$\text{value}(f) = \sum_{(s,j)\in E} f((s, j)) - \sum_{(i,s)\in E} f((i, s)) = \sum_{(i,t)\in E} f((i, t)) - \sum_{(t,j)\in E} f((t, j)).$$

This problem clearly has a linear programming formulation, which we solve below for `s=16` and `t=10`:

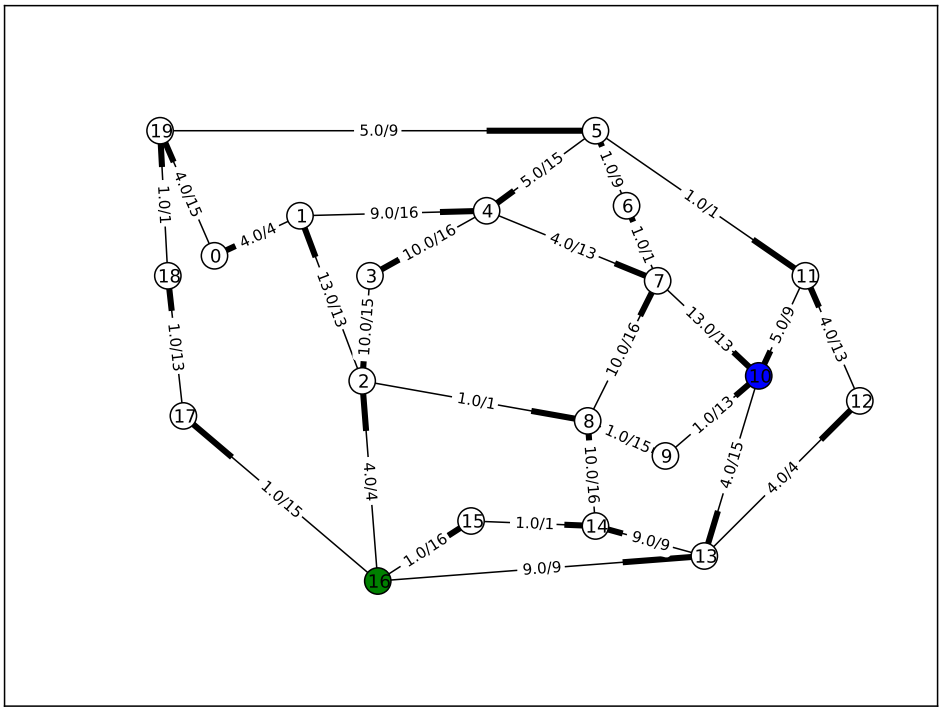
```
maxflow=pic.Problem()
#source and sink nodes
s=16
t=10

#convert the capacities as a picos expression
cc=pic.new_param('c',c)

#flow variable
f={}
for e in G.edges():
        f[e]=maxflow.add_variable('f[{0}]'.format(e),1)


#flow value
F=maxflow.add_variable('F',1)

#upper bound on the flows
maxflow.add_list_of_constraints(
        [f[e]<cc[e] for e in G.edges()], #list of constraints
        [('e',2)],                       #e is a double index
                                         #  (start and end node of the edges)
        'edges'                          #set to which the index e belongs
        )

#flow conservation
maxflow.add_list_of_constraints(
[   pic.sum([f[p,i] for p in G.predecessors(i)],'p','pred(i)')
== pic.sum([f[i,j] for j in G.successors(i)],'j','succ(i)')
for i in G.nodes() if i not in (s,t)],
        'i','nodes-(s,t)')

#source flow at s
maxflow.add_constraint(
pic.sum([f[p,s] for p in G.predecessors(s)],'p','pred(s)') + F
== pic.sum([f[s,j] for j in G.successors(s)],'j','succ(s)')
)

#sink flow at t
maxflow.add_constraint(
pic.sum([f[p,t] for p in G.predecessors(t)],'p','pred(t)')
== pic.sum([f[t,j] for j in G.successors(t)],'j','succ(t)') + F
)

#nonnegativity of the flows
maxflow.add_list_of_constraints(
        [f[e]>0 for e in G.edges()],    #list of constraints
        [('e',2)],                      #e is a double index
                                        #  (origin and destination of the edges)
        'edges'                         #set the index belongs to
        )

#objective
maxflow.set_objective('max',F)

#solve the problem
print maxflow
maxflow.solve(verbose=0)

print 'The optimal flow has value {0}'.format(F)
```

Generated output:

```
---------------------
optimization problem  (LP):
61 variables, 140 affine constraints

f   : dict of 60 variables, (1, 1), continuous
F   : (1, 1), continuous

        maximize F
such that
f[e] < c[e] for all e in edges
Σ_{p in pred(i)} f[(p, i)] = Σ_{j in succ(i)} f[(i, j)] for all i in nodes-(s,t)
Σ_{p in pred(s)} f[(p, 16)] + F = Σ_{j in succ(s)} f[(16, j)]
Σ_{p in pred(t)} f[(p, 10)] = Σ_{j in succ(t)} f[(10, j)] + F
f[e] > 0 for all e in edges
---------------------
The optimal flow has value 15.0
```

Let us now draw the maximum flow:

```
#display the graph
import pylab
fig=pylab.figure(figsize=(11,8))

node_colors=['w']*N
node_colors[s]='g' #source is green
node_colors[t]='b' #sink is blue

pos=nx.spring_layout(G)
#edges
nx.draw_networkx(G,pos,
                 edgelist=[e for e in G.edges() if f[e].value[0]>0],
                 node_color=node_colors)


labels={e:'{0}/{1}'.format(f[e],c[e]) for e in G.edges() if f[e].value[0]>0}
#flow label
nx.draw_networkx_edge_labels(G, pos,
                         edge_labels=labels)

#hide axis
fig.gca().axes.get_xaxis().set_ticks([])
fig.gca().axes.get_yaxis().set_ticks([])

pylab.show()
```



The graph shows the source in blue, the sink in green, and the value of the flow together with the capacity on each edge.

Min-cut

Given a directed graph $G(V, E)$, with a capacity $c(e)$ on each edge $e \in E$, a source node s and a sink node t, the **min-cut** problem is to find a partition of the nodes in two sets (S, T), such that $s \in S$, $t \in T$, and the total capacity of the cut, $\text{capacity}(S, T) = \sum_{(i,j) \in E \cap S \times T} c((i, j))$, is minimized.

It can be seen that binary solutions $d \in \{0, 1\}^E$, $p \in \{0, 1\}^V$ of the following linear program yield a minimum cut:

$$
\begin{array}{ll}
\underset{\substack{d \in \mathbb{R}^E \\ p \in \mathbb{R}^V}}{\text{minimize}} & \displaystyle\sum_{e \in E} c(e) d(e) \\
\text{subject to} & \forall (i, j) \in E,\ d((i, j)) \geq p(i) - p(j) \\
& p(s) = 1 \\
& p(t) = 0 \\
& \forall n \in V,\ p(n) \geq 0 \\
& \forall e \in E,\ d(e) \geq 0
\end{array}
$$

Remarkably, this LP is the dual of the max-flow LP, and the max-flow-min-cut theorem (also known as Ford-Fulkerson theorem *[1]*) states that the capacity of the minimum cut is equal to the value of the maximum flow. This means that the above LP always has an optimal solution in which d is binary. In fact, the matrix defining this LP is *totally unimodular*, from which we know that every extreme point of the polyhedron defining the feasible region is integral, and hence the simplex algorithm will return a minimum cut.

We solve the mincut problem below, for `s=16` and `t=10`:

```
mincut=pic.Problem()

#source and sink nodes
s=16
t=10

#convert the capacities as a picos expression
cc=pic.new_param('c',c)

#cut variable
d={}
for e in G.edges():
        d[e]=mincut.add_variable('d[{0}]'.format(e),1)

#potentials
p=mincut.add_variable('p',N)

#potential inequalities
mincut.add_list_of_constraints(
        [d[i,j] > p[i]-p[j]
        for (i,j) in G.edges()],        #list of constraints
        ['i','j'],'edges')              #indices and set they belong to

#one-potential at source
mincut.add_constraint(p[s]==1)
#zero-potential at sink
mincut.add_constraint(p[t]==0)

#nonnegativity
mincut.add_constraint(p>0)
mincut.add_list_of_constraints(
        [d[e]>0 for e in G.edges()],    #list of constraints
        [('e',2)],                      #e is a double index
                                        #  (origin and destination of the edges)
        'edges'                         #set the index belongs to
        )

#objective
mincut.set_objective('min',
                pic.sum([cc[e]*d[e] for e in G.edges()],
                        [('e',2)],'edges')
                )

print mincut
mincut.solve(verbose=0)

print 'The minimal cut has capacity {0}'.format(mincut.obj_value())

cut=[e for e in G.edges() if d[e].value[0]==1]
S  =[n for n in G.nodes() if p[n].value[0]==1]
T  =[n for n in G.nodes() if p[n].value[0]==0]

print 'the partition of the nodes is: '
print 'S: {0}'.format(S)
print 'T: {0}'.format(T)
```

Generated output:

```
---------------------
optimization problem  (LP):
80 variables, 142 affine constraints

d   : dict of 60 variables, (1, 1), continuous
```

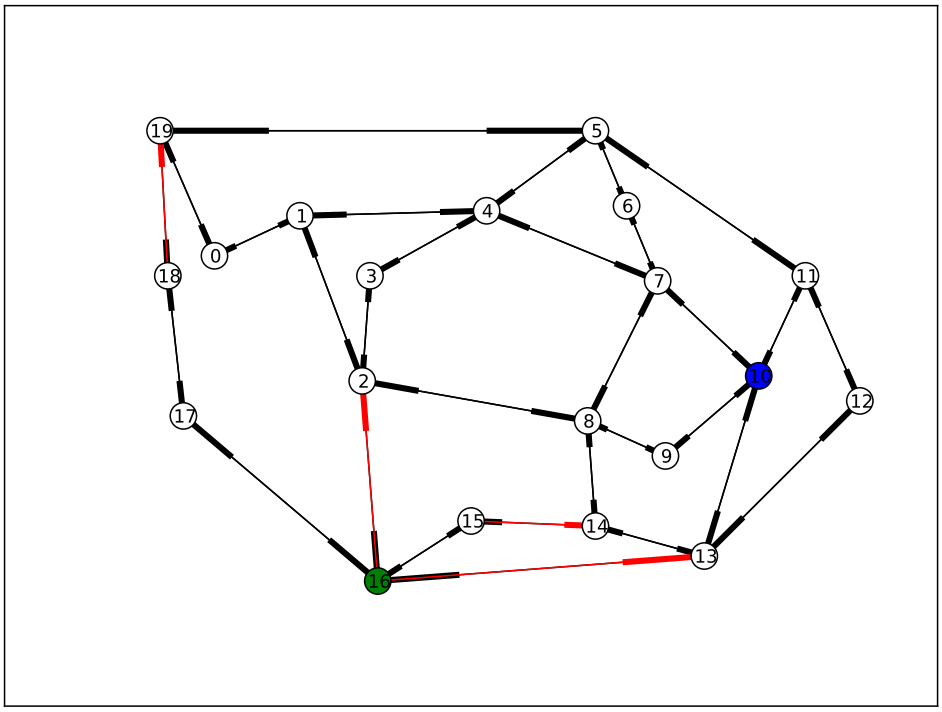
```
p   : (20, 1), continuous

        minimize Σ_{e in edges} c[e]*d[e]
such that
d[(i, j)] > p[i] -p[j] for all (i,j) in edges
p[16] = 1.0
p[10] = 0
p > |0|
d[e] > 0 for all e in edges
---------------------
The minimal cut has capacity 15.0
the partition of the nodes is:
S: [15, 16, 17, 18]
T: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 19]
```

Note that the minimum-cut could also habe been found by using the dual variables of the maxflow LP:

```
>>> #capacited flow constraint
>>> capaflow=maxflow.get_constraint((0,))
>>> dualcut=[e for i,e in enumerate(G.edges()) if capaflow[i].dual[0]==1]
>>> #flow conservation constraint
>>> consflow=maxflow.get_constraint((1,))
>>> Sdual = [s]+ [n for i,n in
...           enumerate([n for n in G.nodes() if n not in (s,t)])
...           if consflow[i].dual[0]==1]
>>> Tdual = [t]+ [n for i,n in
...           enumerate([n for n in G.nodes() if n not in (s,t)])
...           if consflow[i].dual[0]==0]
>>> cut == dualcut
True
>>> set(S) == set(Sdual)
True
>>> set(T) == set(Tdual)
True
```

Let us now draw the maximum flow:

```
import pylab
fig=pylab.figure(figsize=(11,8))

node_colors=['w']*N
node_colors[s]='g' #source is green
node_colors[t]='b' #sink is blue

pos=nx.spring_layout(G)
#edges (not in the cut)
nx.draw_networkx(G,pos,
                 edgelist=[e for e in G.edges() if e not in cut],
                 node_color=node_colors)

#edges of the cut
nx.draw_networkx_edges(G,pos,
                 edgelist=cut,
                 edge_color='r')

#hide axis
fig.gca().axes.get_xaxis().set_ticks([])
fig.gca().axes.get_yaxis().set_ticks([])

pylab.show()
```



On this graph, the source in blue, the sink in green, and the edges defining the cut are marked in red.

3.2.2 Multicut (MIP)

Multicut is a generalization of the mincut problem, in which several pairs of nodes must be disconnected. The goal is to find a cut of minimal capacity, such that for all pair $(s,t) \in \mathcal{P} = \{(s_1,t_1),\ldots,(s_k,t_k))\}$, there is no path from s to t in the graph where the edges of the cut have been removed.

We can obtain a MIP formulation of the multicut problem by doing a small modification the *mincut* LP. The idea is to introduce a different potential for every node which is the source of a pair in $\mathcal{P}$:

$$\forall s \in \mathcal{S} = \{s \in V : \exists t \in V\ (s,t) \in \mathcal{P}\}, p_s \in \mathbb{R}^V,$$

and to force the cut variable to be binary.

$$\begin{array}{ll} \underset{\substack{y\in\{0,1\}^E \\ \forall s\in\mathcal{S},\ p_s\in\mathbb{R}^V}}{\text{minimize}} & \displaystyle\sum_{e\in E} c(e)y(e) \\ \text{subject to} & \forall (i,j), s \in E \times \mathcal{S},\ y((i,j)) \geq p_s(i) - p_s(j) \\ & \forall s \in \mathcal{S},\ p_s(s) = 1 \\ & \forall (s,t) \in \mathcal{P},\ p_s(t) = 0 \\ & \forall (s,n) \in \mathcal{S} \times V,\ p_s(n) \geq 0 \end{array}$$

Unlike the mincut problem, the LP obtained by relaxing the integer constraint $y \in \{0,1\}^E$ is not guaranteed to have an integral solution (see e.g. *[2]*). We solve the multicut problem below, for the terminal pairs $\mathcal{P} = \{(0,12),(1,5),(1,19),(2,11),(3,4),(3,9),(3,18),(6,15),(10,14)\}$.

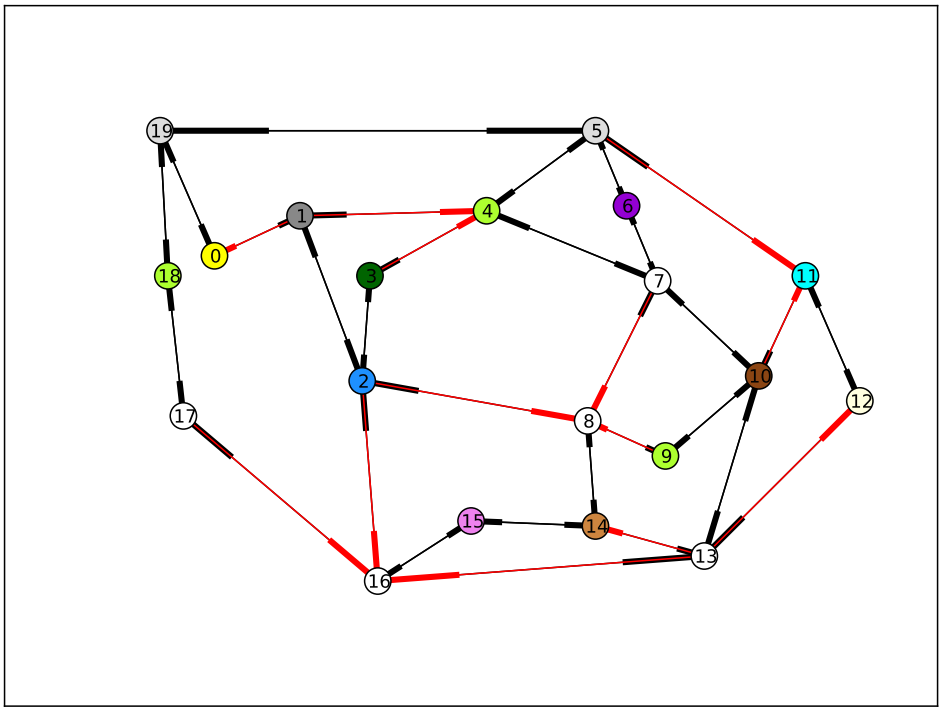
```
multicut=pic.Problem()

#pairs to be separated
pairs=[(0,12),(1,5),(1,19),(2,11),(3,4),(3,9),(3,18),(6,15),(10,14)]

#source and sink nodes
s=16
t=10

#convert the capacities as a picos expression
cc=pic.new_param('c',c)

#list of sources
sources=set([p[0] for p in pairs])


#cut variable
y={}
for e in G.edges():
        y[e]=multicut.add_variable('y[{0}]'.format(e),1,vtype='binary')

#potentials (one for each source)
p={}
for s in sources:
        p[s]=multicut.add_variable('p[{0}]'.format(s),N)

#potential inequalities
multicut.add_list_of_constraints(
        [y[i,j]>p[s][i]-p[s][j]
        for s in sources
        for (i,j) in G.edges()],        #list of constraints
        ['i','j','s'],'edges x sources')#indices and set they belong to

#one-potentials at source
multicut.add_list_of_constraints(
        [p[s][s]==1 for s in sources],
        's','sources')

#zero-potentials at sink
multicut.add_list_of_constraints(
        [p[s][t]==0 for (s,t) in pairs],
        ['s','t'],'pairs')

#nonnegativity
multicut.add_list_of_constraints(
        [p[s]>0 for s in sources],
        's','sources')

#objective
multicut.set_objective('min',
                pic.sum([cc[e]*y[e] for e in G.edges()],
                        [('e',2)],'edges')
                )

print multicut
multicut.solve(verbose=0)

print 'The minimal multicut has capacity {0}'.format(multicut.obj_value())

cut=[e for e in G.edges() if y[e].value[0]==1]

print 'The edges forming the cut are: '
print cut
```

Generated output:

```
---------------------
optimization problem  (MIP):
180 variables, 495 affine constraints

y   : dict of 60 variables, (1, 1), binary
p   : dict of 6 variables, (20, 1), continuous

        minimize Σ_{e in edges} c[e]*y[e]
such that
y[(i, j)] > p[s][i] -p[s][j] for all (i,j,s) in edges x sources
p[s][s] = 1.0 for all s in sources
p[s][t] = 0 for all (s,t) in pairs
p[s] > |0| for all s in sources
---------------------
The minimal multicut has capacity 49.0
The edges forming the cut are:
[(1, 0), (1, 4), (2, 16),
 (2, 8), (3, 4), (5, 11),
 (7, 8), (9, 8), (10, 11),
 (13, 16), (13, 12),
 (13, 14), (17, 16)]
```

Let us now draw the multicut:

```
import pylab

fig=pylab.figure(figsize=(11,8))

#pairs of dark and light colors
colors=[('Yellow','#FFFFE0'),
        ('#888888','#DDDDDD'),
        ('Dodgerblue','Aqua'),
        ('DarkGreen','GreenYellow'),
        ('DarkViolet','Violet'),
        ('SaddleBrown','Peru'),
        ('Red','Tomato'),
        ('DarkGoldenRod','Gold'),
        ]

node_colors=['w']*N
for i,s in enumerate(sources):
        node_colors[s]=colors[i][0]
        for t in [t for (s0,t) in pairs if s0==s]:
                node_colors[t]=colors[i][1]

pos=nx.spring_layout(G)
nx.draw_networkx(G,pos,
                edgelist=[e for e in G.edges() if e not in cut],
                node_color=node_colors)

nx.draw_networkx_edges(G,pos,
                edgelist=cut,
                edge_color='r')

#hide axis
fig.gca().axes.get_xaxis().set_ticks([])
fig.gca().axes.get_yaxis().set_ticks([])

pylab.show()
```



On this graph, the pairs of terminal nodes are denoted by dark and light colors of the same shade (e.g. dark vs. light green for the pairs (3,4),(3,9), and (3,18)), and the edges defining the cut are marked in red.

3.2.3 Maxcut relaxation (SDP)

The goal of the **maxcut** problem is to find a partition (S,T) of the nodes of an *undirected* graph G(V,E), such that the capacity of the cut, $\text{capacity}(S,T) = \sum_{\{i,j\} \in E \cap (S \Delta T)} c((i,j))$, is maximized.

Goemans and Williamson have designed a famous 0.878-approximation algorithm *[3]* for this NP-hard problem based on semidefinite programming. The idea is to introduce a variable $x \in \{-1,1\}^V$ where $x(n)$ takes the value +1 or -1 depending on wheter $n \in S$ or $n \in T$. Then, it can be seen that the value of the cut is equal to $\frac{1}{4} x^T L x$, where L is the Laplacian of the graph. If we define the matrix $X = xx^T$, which is positive semidefinite of rank 1, we obtain an SDP by relaxing the rank-one constraint on X :

$$\begin{array}{ll} \underset{X \in \mathbb{S}_{|V|}}{\text{maximize}} & \frac{1}{4} \langle L, X \rangle \\ \text{subject to} & \text{diag}(X) = \mathbf{1} \\ & X \succeq 0 \end{array}$$

Then, Goemans and Williamson have shown that if we project the solution X onto a random hyperplan, we obtain a cut whose expected capacity is at least 0.878 times the optimum. Below is a simple implementation of their algorithm:

```
import cvxopt as cvx
import cvxopt.lapack
import numpy as np

#make G undirected
G=nx.Graph(G)
```

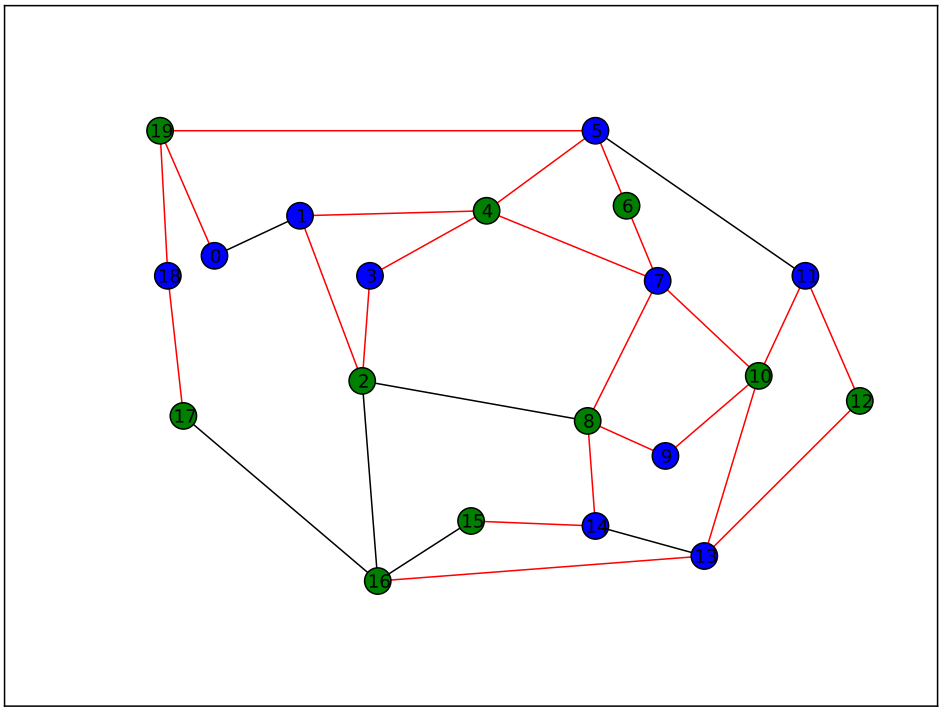
```
#allocate weights to the edges
for (i,j) in G.edges():
        G[i][j]['weight']=c[i,j]+c[j,i]


maxcut = pic.Problem()
X=maxcut.add_variable('X',(N,N),'symmetric')

#Laplacian of the graph
L=pic.new_param('L',1/4.*nx.laplacian(G))

#ones on the diagonal
maxcut.add_constraint(pic.tools.diag_vect(X)==1)
#X positive semidefinite
maxcut.add_constraint(X>>0)

#objective
maxcut.set_objective('max',L|X)

print maxcut
maxcut.solve(verbose = 0)

print 'bound from the SDP relaxation: {0}'.format(maxcut.obj_value())

#---------------------------#
#RANDOM PROJECTION ALGORITHM#
#---------------------------#

#Cholesky factorization
V=X.value

cvxopt.lapack.potrf(V)
for i in range(N):
        for j in range(i+1,N):
                V[i,j]=0

#random projection algorithm
#Repeat 100 times or until we are within a factor .878 of the SDP optimal value
count=0
obj_sdp=maxcut.obj_value()
obj=0
while (count <100 or obj<.878*obj_sdp):
        r=cvx.normal(20,1)
        x=cvx.matrix(np.sign(V*r))
        o=(x.T*L*x).value[0]
        if o>obj:
                x_cut=x
                obj=o
        count+=1

print 'value of the cut: {0}'.format(obj)
S1=[n for n in range(N) if x[n]<0]
S2=[n for n in range(N) if x[n]>0]
cut = [(i,j) for (i,j) in G.edges() if x[i]*x[j]<0]

#we comment this because the output in unpredicatable for doctest:
#print 'partition of the nodes:'
#print 'S1: {0}'.format(S1)
#print 'S2: {0}'.format(S2)
```

Generated output:

```
---------------------
optimization problem  (SDP):
210 variables, 20 affine constraints, 210 vars in 1 SD cones

X   : (20, 20), symmetric

        maximize ⟨ L | X ⟩
such that
diag(X) = |1|
X ≽ |0|
---------------------
bound from the SDP relaxation: 478.2074...
value of the cut: 471.0
```

Let us now draw this cut:

```
#display the cut
import pylab

fig=pylab.figure(figsize=(11,8))

pos=nx.spring_layout(G)

node_colors=[('g' if n in S1 else 'b') for n in range(N)]

nx.draw_networkx(G,pos,
                 edgelist=[e for e in G.edges() if e not in cut],
                 node_color=node_colors)

nx.draw_networkx_edges(G,pos,
                 edgelist=cut,
                 edge_color='r')

#hide axis
fig.gca().axes.get_xaxis().set_ticks([])
fig.gca().axes.get_yaxis().set_ticks([])

pylab.show()
```



On this graph, the red edges are those defining the cut, and the nodes are blue or green depending on the partition they belong to.

3.2.4 References

1. “Maximal Flow through a Network”, LR Ford Jr and DR Fulkerson, *Canadian journal of mathematics*, 1956.
2. “Analysis of LP relaxations for multiway and multicut problems”, D.Bertsimas, C.P. Teo and R. Vohra, *Networks*, 34(2), p. *102-114*, 1999.
3. “Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming”, M.X. Goemans and D.P. Williamson, *Journal of the ACM*, 42(6), p. *1115-1145*, 1995.

CHAPTER

FOUR

THE PICOS API

4.1 Problem

class picos.**Problem**(***options*)

This class represents an optimization problem. The constructor creates an empty problem. Some options can be provided under the form `key = value`. See the list of available options in the doc of `set_all_options_to_default()`

add_constraint(*cons*, *key=None*, *ret=False*)

Adds a constraint in the problem.

Parameters

- **cons** (`Constraint`) – The constraint to be added.
- **key** (*str.*) – Optional parameter to describe the constraint with a key string.
- **ret** (*bool.*) – Do you want the added constraint to be returned ? This can be useful to access the dual of this constraint.

add_list_of_constraints(*lst*, *it=None*, *indices=None*, *key=None*, *ret=False*)

adds a list of constraints in the problem. This fonction can be used with python list comprehensions (see the example below).

Parameters

- **lst** – list of `Constraint`.
- **it** (*None or str or list.*) – Description of the letters which should be used to replace the dummy indices. The function tries to find a template for the string representations of the constraints in the list. If several indices change in the list, their letters should be given as a list of strings, in their order of appearance in the resulting string. For example, if three indices change in the constraints, and you want them to be named `'i'`, `'j'` and `'k'`, set `it = ['i','j','k']`. You can also group two indices which always appear together, e.g. if `'i'` always appear next to `'j'` you could set `it = [('ij',2),'k']`. Here, the number 2 indicates that `'ij'` replaces 2 indices. If `it` is set to `None`, or if the function is not able to find a template, the string of the first constraint will be used for the string representation of the list of constraints.
- **indices** (*str.*) – a string to denote the set where the indices belong to.
- **key** (*str.*) – Optional parameter to describe the list of constraints with a key string.
- **ret** (*bool.*) – Do you want the added list of constraints to be returned ? This can be useful to access the duals of these constraints.

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> prob=pic.Problem()
```

```
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> x
[# variable x[0]:(2 x 1),continuous #,
 # variable x[1]:(2 x 1),continuous #,
 # variable x[2]:(2 x 1),continuous #,
 # variable x[3]:(2 x 1),continuous #,
 # variable x[4]:(2 x 1),continuous #]
>>> y=prob.add_variable('y',5)
>>> IJ=[(1,2),(2,0),(4,2)]
>>> w={}
>>> for ij in IJ:
...         w[ij]=prob.add_variable('w[{0}]'.format(ij),3)
...
>>> u=pic.new_param('u',cvx.matrix([2,5]))
>>> prob.add_list_of_constraints(
... [u.T*x[i]<y[i] for i in range(5)],
... 'i',
... '[5]')
>>>
>>> prob.add_list_of_constraints(
... [abs(w[i,j])<y[j] for (i,j) in IJ],
... [('ij',2)],
... 'IJ')
>>>
>>> prob.add_list_of_constraints(
... [y[t] > y[t+1] for t in range(4)],
... 't',
... '[4]')
>>>
>>> print prob
---------------------
optimization problem (SOCP):
24 variables, 9 affine constraints, 12 vars in 3 SO cones

x   : list of 5 variables, (2, 1), continuous
w   : dict of 3 variables, (3, 1), continuous
y   : (5, 1), continuous

    find vars
such that
  u.T*x[i] < y[i] for all i in [5]
  ||w[ij]|| < y[ij__1] for all ij in IJ
  y[t] > y[t+1] for all t in [4]
---------------------
```

add_variable (*name*, *size=1*, *vtype='continuous'*)

adds a variable in the problem, and returns the corresponding instance of the `Variable`.

For example,

```
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',3)
>>> x
# variable x:(3 x 1),continuous #
```

Parameters

- **name** (*str.*) – The name of the variable.
- **size** (*int or tuple.*) – The size of the variable.

  Can be either:

  – an `int` *n* , in which case the variable is a **vector of dimension n**

  - or a `tuple` *(n,m)*, and the variable is a **n x m-matrix**.
- **vtype** (*str.*) – variable `type`. Can be:
  - `'continuous'` (default),
  - `'binary'`: 0/1 variable
  - `'integer'`: integer valued variable
  - `'symmetric'`: symmetric matrix
  - `'semicont'`: 0 or continuous variable satisfying its bounds
  - `'semiint'`: 0 or integer variable satisfying its bounds

Returns An instance of the class `Variable`.

check_current_value_feasibility(*tol=1e-05*)
returns `True` if the current value of the variabless is a feasible solution, up to the tolerance `tol`. If `tol` is set to `None`, the option parameter `options['tol']` is used instead. The integer feasibility is checked with a tolerance of 1e-3.

constraints = None
list of all constraints

copy()
creates a copy of the problem.

countCons = None
numner of (multidimensional) constraints

countVar = None
number of (multidimensional) variables

get_constraint(*ind*)
returns a constraint of the problem.

Parameters **ind** (*int or tuple.*) – There are two ways to index a constraint.

- if `ind` is an *int* n, then the nth constraint (starting from 0) will be returned, where all the constraints are counted in the order where they were passed to the problem.
- if `ind` is a *tuple* (k, i), then the ith constraint from the kth group of constraints is returned (starting from 0). By *group of constraints*, it is meant a single constraint or a list of constraints added together with the function `add_list_of_constraints()`.
- if `ind` is a tuple of length 1 $(k,)$, then the list of constraints of the kth group is returned.

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> y=prob.add_variable('y',5)
>>> prob.add_list_of_constraints(
... [(1|x[i])<y[i] for i in range(5)],
... 'i',
... '[5]')
>>> prob.add_constraint(y>0)
>>> print prob
---------------------
optimization problem (LP):
15 variables, 10 affine constraints

x   : list of 5 variables, (2, 1), continuous
y   : (5, 1), continuous
```

```
    find vars
  such that
    ⟨ |1| | x[i] ⟩ < y[i] for all i in [5]
    y > |0|
  --------------------
  >>> prob.get_constraint(1)               #2d constraint (numbered from 0)
  # (1x1)-affine constraint: ⟨ |1| | x[1] ⟩ < y[1] #
  >>> prob.get_constraint((0,3))           #4th consraint from the 1st group
  # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #
  >>> prob.get_constraint((1,))            #unique constraint of the 2d 'group'
  # (5x1)-affine constraint: y > |0| #
  >>> prob.get_constraint((0,))            #list of constraints of the 1st group
  [# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
   # (1x1)-affine constraint: ⟨ |1| | x[1] ⟩ < y[1] #,
   # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
   # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
   # (1x1)-affine constraint: ⟨ |1| | x[4] ⟩ < y[4] #]
  >>> prob.get_constraint(5)              #6th constraint
  # (5x1)-affine constraint: y > |0| #
```

get_valued_variable(*name*)

Returns the value of the variable (as an `cvxopt matrix`) with the given `name`. If `name` refers to a list (resp. dict) of variables, named with the template `name[index]` (resp. `name[key]`), then the function returns the list (resp. dict) of these variables.

Parameters **name** (*str.*) – name of the variable, or of a list/dict of variables.

> **Warning:** If the problem has not been solved, or if the variable is not valued, this function will raise an Exception.

get_variable(*name*)

Returns the variable (as a `Variable`) with the given `name`. If `name` refers to a list (resp. dict) of variables, named with the template `name[index]` (resp. `name[key]`), then the function returns the list (resp. dict) of these variables.

Parameters **name** (*str.*) – name of the variable, or of a list/dict of variables.

is_continuous()

Returns `True` if there are only continuous variables

numberAffConstraints = None

total number of (scalar) affine constraints

numberConeConstraints = None

number of SOC constraints

numberConeVars = None

number of auxilary variables required to handle the SOC constraints

numberLSEConstraints = None

number of LogSumExp constraints (+1 if the objective is a LogSumExp)

numberLSEVars = None

number of vars in LogSumExp expressions

numberOfVars = None

total number of (scalar) variables

numberQuadConstraints = None

number of quadratic constraints (+1 if the objective is quadratic)

numberQuadNNZ = None

number of nonzero entries in the matrices defining the quadratic expressions

numberSDPConstraints = None
number of SDP constraints

numberSDPVars = None
size of the s-vecotrized matrices involved in SDP constraints

obj_value()
If the problem was already solved, returns the objective value. Otherwise, it raises an `AttributeError`.

remove_all_constraints()
Removes all constraints from the problem

remove_constraint(*ind*)
Deletes a constraint or a list of constraints of the problem.

Parameters ind (*int or tuple.*) – The indexing of constraints works as in the function `get_constraint()`:

- if `ind` is an integer n, the nth constraint (numbered from 0) is deleted
- if `ind` is a *tuple* (k, i), then the ith constraint from the kth group of constraints is deleted (starting from 0). By *group of constraints*, it is meant a single constraint or a list of constraints added together with the function `add_list_of_constraints()`.
- if `ind` is a tuple of length 1 $(k,)$, then the whole kth group of constraints is deleted.

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(4)]
>>> y=prob.add_variable('y',4)
>>> prob.add_list_of_constraints(
... [(1|x[i])<y[i] for i in range(4)], 'i', '[5]')
>>> prob.add_constraint(y>0)
>>> prob.add_list_of_constraints(
... [x[i]<2 for i in range(3)], 'i', '[3]')
>>> prob.add_constraint(x[3]<1)
>>> prob.constraints
[# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
 # (1x1)-affine constraint: ⟨ |1| | x[1] ⟩ < y[1] #,
 # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
 # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
 # (4x1)-affine constraint: y > |0| #,
 # (2x1)-affine constraint: x[0] < |2.0| #,
 # (2x1)-affine constraint: x[1] < |2.0| #,
 # (2x1)-affine constraint: x[2] < |2.0| #,
 # (2x1)-affine constraint: x[3] < |1| #]
>>> prob.remove_constraint(1)            #2d constraint (numbered from 0) deleted
>>> prob.constraints
[# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
 # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
 # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
 # (4x1)-affine constraint: y > |0| #,
 # (2x1)-affine constraint: x[0] < |2.0| #,
 # (2x1)-affine constraint: x[1] < |2.0| #,
 # (2x1)-affine constraint: x[2] < |2.0| #,
 # (2x1)-affine constraint: x[3] < |1| #]
>>> prob.remove_constraint((1,))         #2d 'group' of constraint deleted,
...                                      #i.e. the single constraint y>|0|
>>> prob.constraints
[# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
 # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
```

```
 # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
 # (2x1)-affine constraint: x[0] < |2.0| #,
 # (2x1)-affine constraint: x[1] < |2.0| #,
 # (2x1)-affine constraint: x[2] < |2.0| #,
 # (2x1)-affine constraint: x[3] < |1| #]
>>> prob.remove_constraint((2,))          #3d 'group' of constraint deleted,
...                                       #(originally the 4th group, i.e. x[3]<|1|)
>>> prob.constraints
[# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
 # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
 # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
 # (2x1)-affine constraint: x[0] < |2.0| #,
 # (2x1)-affine constraint: x[1] < |2.0| #,
 # (2x1)-affine constraint: x[2] < |2.0| #]
>>> prob.remove_constraint((1,1))         #2d constraint of the 2d group (originally
...                                       #the 3rd group, i.e. x[1]<|2| )
>>> prob.constraints
[# (1x1)-affine constraint: ⟨ |1| | x[0] ⟩ < y[0] #,
 # (1x1)-affine constraint: ⟨ |1| | x[2] ⟩ < y[2] #,
 # (1x1)-affine constraint: ⟨ |1| | x[3] ⟩ < y[3] #,
 # (2x1)-affine constraint: x[0] < |2.0| #,
 # (2x1)-affine constraint: x[2] < |2.0| #]
```

remove_variable(*name*)

Removes the variable `name` from the problem. :param name: name of the variable to remove. :type name: str.

set_all_options_to_default()

set all the options to their default. The following options are available:

- General options common to all solvers:
  - `verbose = 1` : verbosity level [0(quiet)|1|2(loud)]
  - `solver = None` : currently the available solvers are `'cvxopt'`, `'cplex'`, `'mosek'`, `'gurobi'`, `'smcp'`, `'zibopt'`. The default `None` means that you let picos select a suitable solver for you.
  - `tol = 1e-8` : Relative gap termination tolerance for interior-point optimizers (feasibility and complementary slackness).
  - `maxit = None` : maximum number of iterations (for simplex or interior-point optimizers). *This option is currently ignored by zibopt*.
  - `lp_root_method = None` : algorithm used to solve continuous LP problems, including the root relaxation of mixed integer problems. The default `None` selects automatically an algorithm. If set to `psimplex` (resp. `dsimplex`, `interior`), the solver will use a primal simplex (resp. dual simplex, interior-point) algorithm. *This option currently works only with cplex, mosek and gurobi*.
  - `lp_node_method = None` : algorithm used to solve subproblems at nodes of the branching trees of mixed integer programs. The default `None` selects automatically an algorithm. If set to `psimplex` (resp. `dsimplex`, `interior`), the solver will use a primal simplex (resp. dual simplex, interior-point) algorithm. *This option currently works only with cplex, mosek and gurobi*.
  - `timelimit = None` : time limit for the solver, in seconds. The default `None` means no time limit. *This option is currently ignored by cvxopt and smcp*.
  - `treememory = None` : size of the buffer for the branch and bound tree, in Megabytes. *This option currently works only with cplex*.
  - `gaplim = 1e-4` : For mixed integer problems, the solver returns a solution as soon as this value for the gap is reached (relative gap between the primal and the dual bound).

– `onlyChangeObjective = False` : set this option to `True` if you have already solved the problem, and want to recompute the solution with a different objective funtion or different parameter settings. This way, the constraints of the problem will not be parsed by picos next time `solve()` is called (this can lead to a huge gain of time).

– `noprimals = False` : if `True`, do not copy the optimal variable values in the `value` attribute of the problem variables.

– `noduals = False` : if `True`, do not try to retrieve the dual variables.

– `nbsol = None` : maximum number of feasible solution nodes visited when solving a mixed integer problem.

– **`hotstart = False`** [if `True`, the MIP optimizer tries to start from the solution] specified (even partly) in the `value` attribute of the problem variables. *This option currently works only with cplex, mosek and gurobi.*

– `convert_quad_to_socp_if_needed = True` : Do we convert the convex quadratics to second order cone constraints when the solver does not handle them directly ?

•Specific options available for cvxopt/smcp:

– `feastol = None` : feasibility tolerance passed to cvx.solvers.options If `feastol` has the default value `None`, then the value of the option `tol` is used.

– `abstol = None` : absolute tolerance passed to cvx.solvers.options If `abstol` has the default value `None`, then the value of the option `tol` is used.

– `reltol = None` : relative tolerance passed to cvx.solvers.options If `reltol` has the default value `None`, then the value of the option `tol`, multiplied by `10`, is used.

•Specific options available for cplex:

– `cplex_params = {}` : a dictionary of cplex parameters to be set before the cplex optimizer is called. For example, `cplex_params={'mip.limits.cutpasses' : 5}` will limit the number of cutting plane passes when solving the root node to `5`.

– `acceptable_gap_at_timelimit = None` : If the the time limit is reached, the optimization process is aborted only if the current gap is less than this value. The default value `None` means that we interrupt the computation regardless of the achieved gap.

•Specific options available for mosek:

– `mosek_params = {}` : a dictionary of mosek parameters to be set before the mosek optimizer is called. For example, `mosek_params={'simplex_abs_tol_piv' : 1e-4}` sets the absolute pivot tolerance of the simplex optimizer to `1e-4`.

•Specific options available for gurobi:

– `gurobi_params = {}` : a dictionary of gurobi parameters to be set before the gurobi optimizer is called. For example, `gurobi_params={'NodeLimit' : 25}` limits the number of nodes visited by the MIP optimizer to 25.

`set_objective`(*typ*, *expr*)

Defines the objective function of the problem.

Parameters

- **typ** (*str.*) – can be either `'max'` (maximization problem), `'min'` (minimization problem), or `'find'` (feasibility problem).
- **expr** – an `Expression`. The expression to be minimized or maximized. This parameter will be ignored if `typ=='find'`.

`set_option`(*key*, *val*)

Sets the option **key** to the value **val**.

Parameters

- **key** (*str.*) – The key of an option (see the list of keys in the doc of `set_all_options_to_default()`).
- **val** – New value for the option.

set_var_value(*name*, *value*, *optimalvar=False*)
Sets the `value` attribute of the given variable.

Parameters

- **name** (*str.*) – name of the variable to which the value will be given
- **value** – The value to be given. The type of `value` must be recognized by the function `_retrieve_matrix()`, so that it can be parsed into a `cvxopt sparse matrix` of the desired size.

Example

```
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',2)
>>> prob.set_var_value('x',[3,4])  #this is equivalent to x.value=[3,4]
>>> abs(x)**2
#quadratic expression: ||x||**2 #
>>> print (abs(x)**2)
25.0
```

solve(***options*)
Solves the problem.

Once the problem has been solved, the optimal variables can be obtained thanks to the property `value` of the class `Expression`. The optimal dual variables can be accessed by the property `dual` of the class `Constraint`.

Parameters **options** – A list of options to update before the call to the solver. In particular, the solver can be specified here, under the form `key = value`. See the list of available options in the doc of `set_all_options_to_default()`

Returns A dictionary **sol** which contains the information returned by the solver.

solver_selection()
Selects an appropriate solver for this problem and sets the option `'solver'`.

status = None
status returned by the solver. The default when a new problem is created is 'unsolved'.

type
Type of Optimization Problem ('LP', 'MIP', 'SOCP', 'QCQP',...)

update_options(***options*)
update the option dictionary, for each pair of the form `key = value`. For a list of available options and their default values, see the doc of `set_all_options_to_default()`.

variables = None
dictionary of variables indexed by variable names

write_to_file(*filename*, *writer='picos'*)
This function writes the problem to a file.

Parameters

- **filename** (*str.*) – The name of the file where the problem will be saved. The extension of the file (if provided) indicates the format of the export:
  - `'.lp'`: LP format . This format handles only linear constraints, unless the writer `'cplex'` is used, and the file is saved in the extended cplex LP format
  - `'.mps'`: MPS format (recquires mosek, gurobi or cplex).
  - `'.opf'`: OPF format (recquires mosek).

- `'.dat-s'`: sparse SDPA format This format is suitable to save semidefinite programs (SDP). SOC constraints are stored as semidefinite constraints with an *arrow pattern*.

- **writer** (*str.*) – The default writer is `picos`, which has its own *LP* and *sparse SDPA* write functions. If cplex, mosek or gurobi is installed, the user can pass the option `writer='cplex'`, `writer='gurobi'` or `writer='mosek'`, and the write function of this solver will be used.

4.2 picos.tools

`picos.tools.`**`available_solvers`**`()`
Lists all available solvers

`picos.tools.`**`diag`**`(`*exp*, *dim=1*`)`
if `exp` is an affine expression of size (n,m), `diag(exp,dim)` returns a diagonal matrix of size `dim*n*m` × `dim*n*m`, with `dim` copies of the vectorized expression `exp[:]` on the diagonal.

In particular:

- when `exp` is scalar, `diag(exp,n)` returns a diagonal matrix of size $n \times n$, with all diagonal elements equal to `exp`.
- when `exp` is a vector of size n, `diag(exp)` returns the diagonal matrix of size $n \times n$ with the vector `exp` on the diagonal

Example

```
>>> import picos as pic
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',1)
>>> y=prob.add_variable('y',1)
>>> pic.tools.diag(x-y,4)
# (4 x 4)-affine expression: Diag(x -y) #
>>> pic.tools.diag(x//y)
# (2 x 2)-affine expression: Diag([x;y]) #
```

`picos.tools.`**`diag_vect`**`(`*exp*`)`
Returns the vector with the diagonal elements of the matrix expression `exp`

Example

```
>>> import picos as pic
>>> prob=pic.Problem()
>>> X=prob.add_variable('X',(3,3))
>>> pic.tools.diag_vect(X)
# (3 x 1)-affine expression: diag(X) #
```

`picos.tools.`**`eval_dict`**`(`*dict_of_variables*`)`
if `dict_of_variables` is a dictionary mapping variable names (strings) to `variables`, this function returns the dictionary `names -> variable values`.

`picos.tools.`**`lse`**`(`*exp*`)`
shorter name for the constructor of the class `LogSumExp`

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',3)
>>> A=pic.new_param('A',cvx.matrix([[1,2],[3,4],[5,6]]))
>>> pic.lse(A*x)<0
# (2x1)-Geometric Programming constraint LSE[ A*x ] < 0 #
```

picos.tools.**new_param**(*name*, *value*)

Declare a parameter for the problem, that will be stored as a cvxopt sparse matrix. It is possible to give a list or a dictionary of parameters. The function returns a constant AffinExp (or a list or a dict of AffinExp) representing this parameter.

Note: Declaring parameters is optional, since the expression can as well be given by using normal variables. (see Example below). However, if you use this function to declare your parameters, the names of the parameters will be displayed when you **print** an Expression or a Constraint

---

Parameters

- **name** (*str.*) – The name given to this parameter.
- **value** – The value (resp list of values, dict of values) of the parameter. The type of **value** (resp. the elements of the list **value**, the values of the dict **value**) should be understandable by the function _retrieve_matrix().

Returns A constant affine expression (AffinExp) (resp. a list of AffinExp of the same length as **value**, a dict of AffinExp indexed by the keys of **value**)

Example:

```
>>> import cvxopt as cvx
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',3)
>>> B={'foo':17.4,'matrix':cvx.matrix([[1,2],[3,4],[5,6]]),'ones':'|1|(4,1)'}
>>> B['matrix']*x+B['foo']
# (2 x 1)-affine expression: [ 2 x 3 MAT ]*x + |17.4| #
>>> #(in the string above, |17.4| represents the 2-dim vector [17.4,17.4])
>>> B=pic.new_param('B',B)
>>> #now that B is a param, we have a nicer display:
>>> B['matrix']*x+B['foo']
# (2 x 1)-affine expression: B[matrix]*x + |B[foo]| #
```

picos.tools.**sum**(*lst*, *it=None*, *indices=None*)

sum of a list of affine expressions. This fonction can be used with python list comprehensions (see the example below).

Parameters

- **lst** – list of AffinExp.
- **it** (*None or str or list.*) – Description of the letters which should be used to replace the dummy indices. The function tries to find a template for the string representations of the affine expressions in the list. If several indices change in the list, their letters should be given as a list of strings, in their order of appearance in the resulting string. For example, if three indices change in the summands, and you want them to be named 'i', 'j' and 'k', set it = ['i','j','k']. You can also group two indices which always appear together, e.g. if 'i' always appear next to 'j' you could set it = [('ij',2),'k']. Here, the number 2 indicates that 'ij' replaces 2 indices. If it is set to None, or if the function is not able to find a template, the string of the first summand will be used for the string representation of the sum.
- **indices** (*str.*) – a string to denote the set where the indices belong to.

Example:

```
>>> import picos as pic
>>> prob=pic.Problem()
>>> x={}
>>> names=['foo','bar','baz']
>>> for n in names:
...   x[n]=prob.add_variable( 'x[{0}]'.format(n),(3,5) )
```

```
>>> x
{'baz': # variable x[baz]:(3 x 5),continuous #,
 'foo': # variable x[foo]:(3 x 5),continuous #,
 'bar': # variable x[bar]:(3 x 5),continuous #}
>>> pic.sum([x[n] for n in names],'n','names')
# (3 x 5)-affine expression: Σ_{n in names} x[n] #
>>> pic.sum([(i+1) * x[n] for i,n in enumerate(names)],['i','n'],'[3] x names') #two indices
# (3 x 5)-affine expression: Σ_{i,n in [3] x names} i*x[n] #
>>> IJ = [(1,2),(2,4),(0,1),(1,3)]
>>> pic.sum([x['foo'][ij] for ij in IJ],[('ij',2)],'IJ') #double index
# (1 x 1)-affine expression: Σ_{ij in IJ} x[foo][ij] #
```

picos.tools.**_retrieve_matrix**(*mat*, *exSize=None*)

parses the variable *mat* and convert it to a `cvxopt sparse matrix`. If the variable **exSize** is provided, the function tries to return a matrix that matches this expected size, or raise an error.

> **Warning:** If there is a conflit between the size of **mat** and the expected size **exsize**, the function might still return something without raising an error !

Parameters **mat** – The value to be converted into a cvx.spmatrix. The function will try to parse this variable and format it to a vector/matrix. *mat* can be of one of the following types:

- `list` [creates a vecor of dimension len(list)]
- `cvxopt matrix`
- `cvxopt sparse matrix`
- `numpy array`
- `int` or `real` [creates a vector/matrix of the size exSize *(or of size (1,1) if exSize is None)*, whith all entries equal to **mat**.
- following strings:
  - '`|a|`' for a matrix with all terms equal to a
  - '`|a|(n,m)`' for a matrix forced to be of size n x m, with all terms equal to a
  - '`e_i(n,m)`' matrix of size (n,m), with a 1 on the ith coordinate (and 0 elsewhere)
  - '`e_i,j(n,m)`' matrix of size (n,m), with a 1 on the (i,j)-entry (and 0 elsewhere)
  - '`I`' for the identity matrix
  - '`I(n)`' for the identity matrix, forced to be of size n x n.
  - '`a%s`', where `%s` is one of the above string: the matrix that should be returned when **mat** == `%s`, multiplied by the scalar a.

Returns A tuple of the form (**M**, **s**), where **M** is the conversion of **mat** into a `cvxopt sparse matrix`, and **s** is a string representation of **mat**

Example:

```
>>> import picos as pic
>>> pic.tools._retrieve_matrix([1,2,3])
(<3x1 sparse matrix, tc='d', nnz=3>, '[ 3 x 1 MAT ]')
>>> pic.tools._retrieve_matrix('e_5(7,1)')
(<7x1 sparse matrix, tc='d', nnz=1>, 'e_5')
>>> print pic.tools._retrieve_matrix('e_11(7,2)')[0]
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        1.00e+00]
[   0        0       ]
[   0        0       ]
```

```
>>> print pic.tools._retrieve_matrix('5.3I',(2,2))
(<2x2 sparse matrix, tc='d', nnz=2>, '5.3I')
```

4.3 Expression

class picos.**Expression**(*string*)

The parent class of AffinExp (which is the parent class of Variable), Norm, LogSumExp, and QuadExp.

4.3.1 AffinExp

class picos.**AffinExp**(*factors=None*, *constant=None*, *size=(1, 1)*, *string='0'*)

A class for defining vectorial (or matrix) affine expressions. It derives from Expression.

Overloaded operators

- **+** sum (with an affine or quadratic expression)
- **+=** in-place sum (with an affine or quadratic expression)
- **-** substraction (with an affine or quadratic expression) or unitary minus
- ***** multiplication (by another affine expression or a scalar)
- **/** division (by a scalar)
- **|** scalar product (with another affine expression)
- **[.]** slice of an affine expression
- **abs()** Euclidean norm (Frobenius norm for matrices)
- ****** exponentiation (works with arbitrary powers for constant affine expressions, and only with the exponent 2 if the affine expression involves some variables)
- **&** horizontal concatenation (with another affine expression)
- **//** vertical concatenation (with another affine expression)
- **<** less **or equal** (than an affine or quadratic expression)
- **>** greater **or equal** (than an affine or quadratic expression)
- **==** is equal (to another affine expression)
- **<<** less than inequality in the Loewner ordering (linear matrix inequality $\preceq$)
- **>>** greater than inequality in the Loewner ordering (linear matrix inequality $\succeq$)

Warning: We recall here the implicit assumptions that are made when using relation operator overloads, in the following two situations:

- the rotated second order cone constraint `abs(exp1)**2 < exp2 * exp3` implicitely assumes that the scalar expression `exp2` (and hence `exp3`) **is nonnegative**.
- the linear matrix inequality `exp1 >> exp2` only tells picos that the symmetric matrix whose lower triangular elements are those of `exp1-exp2` is positive semidefinite. The matrix `exp1-exp2` **is not constrained to be symmetric**. Hence, you should manually add the constraint `exp1-exp2 == (exp1-exp2).T` if it is not clear from the data that this matrix is symmetric.

T

Transposition

constant = None

constant of the affine expression, stored as a cvxopt sparse matrix.

factors = None
dictionary storing the matrix of coefficients of the linear part of the affine expressions. The matrices of coefficients are always stored with respect to vectorized variables (for the case of matrix variables), and are indexed by instances of the class `Variable`.

is0()
is the expression equal to 0 ?

isconstant()
is the expression constant (no variable involved) ?

size = None
size of the affine expression

value
value of the affine expression

Variable

class picos.**Variable**(*name*, *size*, *Id*, *startIndex*, *vtype='continuous'*)
This class stores a variable. It derives from `AffinExp`.

Id = None
An integer index

endIndex = None
end position in the global vector of all variables

name = None
The name of the variable (str)

startIndex = None
starting position in the global vector of all variables

value
value of the variable. The value of a variable is defined in the following two situations:

- The user has assigned a value to the variable, by using either the present `value` attribute, or the function `set_var_value()` of the class `Problem`.
- The problem involving the variable has been solved, and the `value` attribute stores the optimal value of this variable.

vtype = None
one of the following strings:

- 'continuous' (continuous variable)
- 'binary' (binary 0/1 variable)
- 'integer' (integer variable)
- 'symmetric' (symmetric matrix variable)
- **'semicont' (semicontinuous variable** [can take the value 0 or any other admissible value])
- **'semiint' (semi integer variable** [can take the value 0 or any other integer admissible value])

4.3.2 Norm

class picos.**Norm**(*exp*)
Euclidean (or Frobenius) norm of an Affine Expression. This class derives from `Expression`.

Overloaded operators

`**` exponentiation (only implemented when the exponent is 2)

< less **or equal** (than a scalar affine expression)

exp = None
The affine expression of which we take the norm

4.3.3 QuadExp

class picos.**QuadExp**(*quad*, *aff*, *string*, *LR=None*)
Quadratic expression. This class derives from Expression.

Overloaded operators

+ addition (with an affine or a quadratic expression)

− substraction (with an affine or a quadratic expression) or unitary minus

***** multiplication (by a scalar or a constant affine expression)

< less **or equal** than (another quadratic or affine expression).

> greater **or equal** than (another quadratic or affine expression).

LR = None
stores a factorization of the quadratic expression, if the expression was entered in a factorized form. We have:

- `LR=None` when no factorization is known
- `LR=(aff,None)` when the expression is equal to `||aff||**2`
- `LR=(aff1,aff2)` when the expression is equal to `aff1*aff2`.

aff = None
affine expression representing the affine part of the quadratic expression

quad = None
dictionary of quadratic forms, stored as matrices representing bilinear forms with respect to two vectorized variables, and indexed by tuples of instances of the class Variable.

4.3.4 LogSumExp

class picos.**LogSumExp**(*exp*)

Log-Sum-Exp applied to an affine expression. If the affine expression z is of size N, with elements $z_1, z_2, \ldots, z_N$, then LogSumExp(z) represents the expression $\log(\sum_{i=1}^{n} e^{z_i})$. This class derives from Expression.

Overloaded operator

< less **or equal** than (the rhs **must be 0**, for geometrical programming)

value
value of the logsumexp expression

4.4 Constraint

class picos.**Constraint**(*typeOfConstraint*, *Id*, *Exp1*, *Exp2*, *Exp3=None*, *dualVariable=None*, *key=None*)
A class for describing a constraint.

Exp1 = None
LHS

Exp2 = None
RHS (ignored for constraints of type `lse` and `quad`, where `Exp2` is set to `0`)

Exp3 = None
Second factor of the RHS for `RScone` constraints (see `typeOfConstraint`).

Id = None
An integer identifier

dual
Value of the dual variable associated to this constraint

See the *note on dual variables* in the tutorial for more information.

key = None
A string to give a key name to the constraint

slack
Value of the slack variable associated to this constraint (should be nonnegative/zero if the inequality/equality is satisfied)

slack_var()
returns the slack of the constraint (For an inequality of the type `lhs<rhs`, the slack is `rhs-lhs`, and for `lhs>rhs` the slack is `lhs-rhs`).

typeOfConstraint = None
A string from the following values, indicating the type of constraint:

- `lin<`, `lin=` or `lin>` : Linear (in)equality `Exp1` $\leq$ `Exp2`, `Exp1 = Exp2` or `Exp1` $\geq$ `Exp2`.
- `SOcone` : Second Order Cone constraint `||Exp1|| < Exp2`.
- `RScone` : Rotated Cone constraint `||Exp1||**2 < Exp2 * Exp3`.
- `lse` : Geometric Programming constraint `LogSumExp(Exp1)<0`
- `quad`: scalar quadratic constraint `Exp1 < 0`.
- `sdp<` or `sdp>`: semidefinite constraint `Exp1` $\preceq$ `Exp2` or `Exp1` $\succeq$ `Exp2`.

INDEX