PICOS

Full Documentation

Release 1.2.0

Guillaume Sagnol **Maximilian Stahlberg**

2019-01-18

Contents

CHAPTER 1

A Python Interface to Conic Optimization Solvers

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

PICOS allows you to enter a mathematical optimization problem as a **high level model**, with painless support for **(complex) vector and matrix variables** and **multidemensional algebra**. Your model will be transformed to the standard form understood by an appropriate solver that is available at runtime. This makes your application **portable** as users have the choice between several commercial and open source solvers.

1.1 Features

PICOS runs under both **Python 2** and **Python 3** and supports the following solvers and problem types. To use a solver, you need to seperately install it along with the Python interface listed here.

Solver	Interface	LP	SOCP	SDP	QCQP	EXP	MIP	License
CPLEX	included	Yes	Yes		Yes		Yes	non-free
CVXOPT	native	Yes	Yes	Yes	Yes	Yes[1]		GPL-3
ECOS	ecos-python	Yes	Yes		Yes	Yes	Yes	GPL-3
GLPK	swiglpk	Yes					Yes	GPL-3
Gurobi	included	Yes	Yes		Yes		Yes	non-free
MOSEK	included	Yes	Yes	Yes	Yes		Yes	non-free
SMCP	native	Yes[2]	Yes[2]	Yes	Yes[2]			GPL-3
SCIP	PySCIPOpt	Yes	Yes		Yes		Yes	ZIB/MIT

[1] only geometric programming, [2] experimental

1.1.1 Example

This is what it looks like to solve a multidimensional mixed integer program with PICOS:

```
>>> import picos
>>> P = picos.Problem()
>>> x = P.add_variable("x", 2, vtype="integer")
>>> C = P.add_constraint(x <= 5.5)
>>> P.set_objective("max", 1|x) # 1|x is the sum over x
>>> solution = P.solve(verbose = 0)
```

(continues on next page)

(continued from previous page)

```
>>> print(solution["status"])
'integer optimal solution'
>>> print(P.obj_value())
10.0
>>> print(x)
[ 5.00e+00]
[ 5.00e+00]
>>> print(C.slack)
[ 5.00e-01]
[ 5.00e-01]
```

1.1.2 Documentation & Source

- The full documentation can be browsed online or downloaded in PDF form.
- The API documentation without the tutorial and examples is also available as a separate PDF.
- The source code lives on GitLab.

1.2 Installation

1.2.1 Via pip

If you are using pip you can run `pip install picos` to get the latest version.

1.2.2 Via Anaconda

If you are using Anaconda you can run `conda install -c picos picos` to get the latest version.

1.2.3 Via your system's package manager

On **Arch Linux**, there are seperate packages in the AUR for the latest version and the latest release. Both are split packages that ship both Python 2 and Python 3 versions of PICOS.

If you are packaging PICOS for additional systems, please tell us so we can list your package here!

1.2.4 From source

If you are installing PICOS manually, you can choose between a number of development versions and source releases. You will need to have at least the following Python packages installed:

- NumPy
- CVXOPT

1.3 Credits

1.3.1 Developers

- Guillaume Sagnol is PICOS' initial author and primary developer since 2012.
- Maximilian Stahlberg is extending and maintaining PICOS since 2017.

1.3.2 Contributors

For an up-to-date list of all code contributors, please refer to the contributors page. Should a reference from before 2019 be unclear, you can refer to the old contributors page on GitHub as well.

1.4 License

PICOS is free and open source software and available to you under the terms of the GNU GPL v3.

CHAPTER 2

User's Guide

If you are new to PICOS, you can start with the *tutorial* or dig into our *examples*. As an experienced user, you can refer to the *function cheat sheet* or go directly to the *API reference*.

2.1 Tutorial

First of all, let us import PICOS, CVXOPT, and a pretty printing helper

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

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

Let us now create an instance `prob` of an optimization problem

```
>>> prob = pic.Problem()
```

2.1.1 Output settings

PICOS makes heavy use of unicode symbols to generate pretty output. If you find that some of these symbols are not available on your terminal, you can call *`picos.ascii()`* or *`picos.latin1()`* to restrict the charset used.

```
>>> # Create a dummy symmetric matrix variable to showcase different charsets.
>>> X = pic.Problem().add_variable('X', (2,2), vtype="symmetric")
>>> # Show the default text representation of the SDP constraint X ≽ 0.
>>> print(X >> 0)
X ≽ 0
>>> # Restrict to ISO 8859-1 (aka Latin-1).
>>> pic.latin1()
>>> print(X >> 0)
X » 0
>>> # Restrict to pure ASCII.
>>> pic.ascii()
>>> print(X >> 0)
X >> 0
```

In addition to complete charset changes, PICOS allows you to fine-tune how certain expressions are formatted. See *picos.glyphs* for more.

```
>>> pic.glyphs.psdge.template = "{} - {} is positive semidefinite"
>>> print(X >> 0)
X - 0 is positive semidefinite
```

For the sake of this tutorial, we return to the full beauty of unicode.

```
>>> pic.default_charset() # Revert the changes; the same as pic.unicode().
```

2.1.2 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]
<1×1 Binary Variable: w[(2, 4)]>
>>> Y
<2×4 Continuous Variable: Y>
```

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

```
>>> w[2,4].vtype
'binary'
>>> x.vtype
'continuous'
>>> x.vtype='integer'
>>> x
<4×1 Integer Variable: x>
>>> x.size
```

(continues on next page)

(continued from previous page)

```
(4, 1)
>>> Z[1].value = A[0].T
>>> Z[0].is_valued()
False
>>> Z[1].is_valued()
True
>>> Z[2].name
'Z[2]'
```

The admissible values for the `vtype` attribute are documented in *add_variable()*.

2.1.3 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]
<4×2 Affine Expression: Z[0] + Z[3]>
```

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

```
>>> x
<4×1 Integer Variable: x>
>>> x.T
<1×4 Affine Expression: x^T>
```

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]
<4×1 Affine Expression: x + [4×1]>
>>> x.T + b[0]
<1×4 Affine Expression: x^T + [1×4]>
>>> A[0] * Z[0] + A[4] * Z[4]
<2×2 Affine Expression: [2×4]·Z[0] + [2×4]·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)              #this creates a list of constant affine␣
↪expressions [A[0],...,A[4]]
>>> b = pic.new_param('b',b)              #this creates a list of constant affine␣
↪expressions [b[0],...,b[4]]
>>> D = pic.new_param('D',D)              #this creates a dictionary of constant␣
↪AffExpr, indexed by 'Peter', 'Bob', ...
>>> alpha = pic.new_param('alpha',12)     #a scalar parameter
```

```
>>> alpha
<1×1 Affine Expression: alpha>
>>> D['Betty']
<1×1 Affine Expression: D[Betty]>
>>> pprint(b)
```

(continues on next page)

(continued from previous page)

```
[<4×1 Affine Expression: b[0]>,
 <4×1 Affine Expression: b[1]>,
 <4×1 Affine Expression: b[2]>,
 <4×1 Affine Expression: b[3]>,
 <4×1 Affine Expression: b[4]>]
>>> print(b[0])
[ 0.00e+00]
[ 2.00e+00]
[ 0.00e+00]
[ 3.00e+00]
<BLANKLINE>
```

The above example also illustrates that when a *valued* affine expression is printed, then its value is displayed instead of a symbolic description. 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
>>> x_minus_1                          #note that 1 was recognized as the (4x1)-
↪vector with all ones
<4×1 Affine Expression: x - [1]>
>>> print(x_minus_1)
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]
<BLANKLINE>
```

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])                          # [1,2,3] is converted␣
↪into a vector of dimension 3
<3×1 Affine Expression: vect>
>>> pic.new_param('mat',[[1,2,3],[4,5,6]])                 # [[1,2,3],[4,5,6]] is␣
↪converted into a (2x3)-matrix
<2×3 Affine Expression: mat>
>>> pic.new_param('list_of_scalars',(1,2,3))               # (1,2,3) is converted␣
↪into a list of 3 scalar parameters #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Expression: list_of_scalars[0]>,
 <1×1 Affine Expression: list_of_scalars[1]>,
 <1×1 Affine Expression: list_of_scalars[2]>]
>>> pic.new_param('list_of_vectors',([1,2,3],[4,5,6]))     # ([1,2,3],[4,5,6]) is␣
↪converted into a list of 2 vector parameters #doctest: +NORMALIZE_WHITESPACE
[<3×1 Affine Expression: list_of_vectors[0]>,
 <3×1 Affine Expression: list_of_vectors[1]>]
```

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
<2×2 Affine Expression: A[0]·Z[0]>
>>> Z[0] * A[0]                                  #right multiplication
<4×4 Affine Expression: Z[0]·A[0]>
>>> A[1] * Z[0] * A[2]                           #left and right multiplication
<2×4 Affine Expression: A[1]·Z[0]·A[2]>
>>> alpha*Y                                      #scalar multiplication
<2×4 Affine Expression: alpha·Y>
>>> t/b[1][3] - D['Bob']                         #division by a scalar and substraction
<1×1 Affine Expression: t/b[1][3] - D[Bob]>
>>> ( b[2] | x )                                 #dot product
<1×1 Affine Expression: ⟨b[2], x⟩>
>>> ( A[3] | Y )                                 #generalized dot product for␣
↪matrices: (A|B)=trace(A*B.T)
<1×1 Affine Expression: ⟨A[3], Y⟩>
>>> b[1]^x                                       #hadamard (element-wise) product
<4×1 Affine Expression: b[1]⊙x>
```

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]
<2×1 Affine Expression: b[1][1:3]>
>>> Y[1,:]                                       #2d row of Y
<1×4 Affine Expression: Y[1,:]>
>>> x[-1]                                        #last element of x
<1×1 Affine Expression: x[-1]>
>>> A[2][:,1:3]*Y[:,-2::-2]                      #extended slicing with (negative)␣
↪steps is allowed
<2×2 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 `//` (vertical concatenation) and `&` (horizontal concatenation) have been overloaded:

```
>>> (b[1] & b[2] & x & A[0].T*A[0]*x) // x.T
<5×4 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
<4×1 Affine Expression: 5·x - [alpha]>
```

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)])
<2×2 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)])
<2×2 Affine Expression: ∑(A[i]·Z[i] : i ∈ [0...4])>
```

2.1.4 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) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (Mixed-Integer Unconstrained Problem):
59 variables, 0 affine constraints
<BLANKLINE>
Y    : (2, 4), continuous
Z    : list of 5 variables, (4, 2), continuous
t    : (1, 1), continuous
w    : dict of 6 variables, (1, 1), binary
x    : (4, 1), integer
<BLANKLINE>
    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).

2.1.5 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 4×1 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 4×2 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 1×1 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.

It is also possible to use other L_p- norms in picos, cf. *this paragraph* .

2.1.6 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: t² - x[1]·x[2] + 2·t - alpha>
>>> (x[1]-2) * (t+4)                                  #product of two affine␣
↪expressions
<Quadratic Expression: (x[1] - 2)·(t + 4)>
>>> Y[0,:]*x                                          #Row vector multiplied by␣
↪column vector
<Quadratic Expression: Y[0,:]·x>
>>> (x +2 | Z[1][:,1])                                #scalar product of affine␣
↪expressions
<Quadratic Expression: ⟨x + [2], Z[1][:,1]⟩>
>>> abs(x)**2                                         #recall that abs(x) is␣
↪the euclidean norm of x
```

(continues on next page)

(continued from previous page)

```
<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.

2.1.7 Constraints

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

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] less or equal than 2
<1×1 Affine Constraint: ⟨[1], x⟩ ≤ 2>
>>> Z[0] * A[0] > b[1]*b[2].T                                  #A 4x4-
↪elementwise inequality
<4×4 Affine Constraint: Z[0]·A[0] ≥ b[1]·b[2]ᵀ>
>>> pic.sum([A[i]*Z[i] for i in range(5)]) == 0            #A 2x2 equality. The␣
↪RHS is the all-zero matrix
<2×2 Affine Constraint: ∑(A[i]·Z[i] : i ∈ [0...4]) = 0>
```

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

```
>>> constraints = []
>>> for i in range(1,5):
...     constraints.append(prob.add_constraint(Z[i]==Z[i-1]+Y.T))
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (MILP):
59 variables, 32 affine constraints
<BLANKLINE>
Y   : (2, 4), continuous
Z   : list of 5 variables, (4, 2), continuous
t   : (1, 1), continuous
w   : dict of 6 variables, (1, 1), binary
x   : (4, 1), integer
<BLANKLINE>
    maximize ⟨A[0], Y⟩ - t
such that
  Z[1] = Z[0] + Yᵀ
  Z[2] = Z[1] + Yᵀ
  Z[3] = Z[2] + Yᵀ
  Z[4] = Z[3] + Yᵀ
---------------------
```

The constraints of the problem can be accessed via the reference returned by *add_constraint()* or with the function *get_constraint()*:

```
>>> prob.get_constraint(2)
<4×2 Affine Constraint: Z[3] = Z[2] + Yᵀ>
>>> constraints[2] is prob.get_constraint(2)
True
```

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()*.

```
>>> # We first remove all constraints previously added.
>>> prob.remove_all_constraints()
>>> # Then we add a single constraint.
>>> C = prob.add_constraint(Y > 0)
>>> # And then the same list of constraints as above.
>>> CL = prob.add_list_of_constraints(
...      [Z[i] == Z[i-1] + Y.T for i in range(1,5)])
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (MILP):
59 variables, 40 affine constraints
<BLANKLINE>
Y    : (2, 4), continuous
Z    : list of 5 variables, (4, 2), continuous
t    : (1, 1), continuous
w    : dict of 6 variables, (1, 1), binary
x    : (4, 1), integer
<BLANKLINE>
    maximize ⟨A[0], Y⟩ - t
such that
  Y ≥ 0
  Z[i+1] = Z[i] + Y^T ∀ i ∈ [0...3]
---------------------
```

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))
<4×2 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).

Removing constraints

It can be useful to remove some constraints, especially for dynamic approaches such as column generation. Re-creating an instance from scratch after each iteration would be inefficient. Instead, PICOS allows one to modify the solver instance and to re-solve it on the fly (for `mosek`, `cplex` and `gurobi`). It is not possible to change directly a constraint, but you can delete a constraint from model, and then re-add the modified constraint.

The next code shows an example in which a variable `x2` is added to the model, which appears as `+3*x2` in the objective function and as `+x2` in the LHS of a constraint.

```
>>> prb = pic.Problem()
>>> x1 = prb.add_variable('x1',1)
>>> lhs = 2*x1
>>> obj = 5*x1
>>> cons = prb.add_constraint(lhs <= 1)
>>> sol = prb.maximize(obj,verbose=0,solver='cvxopt')
>>> #---------------------------------------
>>> #at this place, the user can use his favorite method to solve the 'pricing␣
↪problem'.
>>> #Let us assume that this phase suggests to add a new variable 'x2' in the␣
↪model,
>>> #which appears as `+3*x2` in the objective function, and as `+x2` in the LHS␣
↪of the constraint.
>>> x2 = prb.add_variable('x2',1)
>>> lhs += x2
>>> obj += 3*x2
```

(continues on next page)

(continued from previous page)

```
>>> cons.delete()
>>> newcons = prb.add_constraint(lhs <= 1)
>>> print(prb) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (LP):
2 variables, 1 affine constraints
<BLANKLINE>
x1  : (1, 1), continuous
x2  : (1, 1), continuous
<BLANKLINE>
    maximize 5·x1 + 3·x2
such that
  2·x1 + x2 ≤ 1
---------------------
```

Flow constraints in Graphs

Flow constraints in graphs are entered using a Networkx Graph. The following example finds a (trivial) maximal flow from `'S'` to `'T'` in `G`.

```
>>> import networkx as nx
>>> G = nx.DiGraph()
>>> G.add_edge('S','A', capacity=1)
>>> G.add_edge('A','B', capacity=1)
>>> G.add_edge('B','T', capacity=1)
>>> pb = pic.Problem()
>>> # Adding the flow variables
>>> f={}
>>> for e in G.edges():
...     f[e]=pb.add_variable('f[{0}]'.format(e),1)
>>> # A variable for the value of the flow
>>> F = pb.add_variable('F',1)
>>> # Creating the flow constraint
>>> flowCons = pb.add_constraint(pic.flow_Constraint(
...     G, f, 'S', 'T', F, capacity='capacity', graphName='G'))
>>> pb.set_objective('max',F)
>>> sol = pb.solve(verbose=0,solver='cvxopt')
>>> flow = pic.tools.eval_dict(f)
```

Picos allows you to define single source - multiple sinks problems. You can use the same syntax as for a single source - single sink problems. Just add a list of sinks and a list of flows instead.

```
import picos as pic
import networkx as nx

G=nx.DiGraph()
G.add_edge('S','A', capacity=2); G.add_edge('S','B', capacity=2)
G.add_edge('A','T1', capacity=2); G.add_edge('B','T2', capacity=2)

pbMultipleSinks=pic.Problem()
# Flow variable
f={}
for e in G.edges():
    f[e]=pbMultipleSinks.add_variable('f[{0}]'.format(e),1)

# Flow value
F1=pbMultipleSinks.add_variable('F1',1)
F2=pbMultipleSinks.add_variable('F2',1)

flowCons = pic.flow_Constraint(
    G, f, source='S', sink=['T1','T2'], capacity='capacity',
```

(continues on next page)

(continued from previous page)

```
    flow_value=[F1, F2], graphName='G')

pbMultipleSinks.add_constraint(flowCons)
pbMultipleSinks.set_objective('max',F1+F2)

# Solve the problem
pbMultipleSinks.solve(verbose=0,solver='cvxopt')

print(pbMultipleSinks)
print('The optimal flow F1 has value {:.1f}'.format(F1.value))
print('The optimal flow F2 has value {:.1f}'.format(F2.value))
```

```
---------------------
optimization problem (LP):
6 variables, 12 affine constraints

F1       : (1, 1), continuous
F2       : (1, 1), continuous
f        : dict of 4 variables, (1, 1), continuous

          maximize F1 + F2
such that
  Feasible S-(T1, T2)-flow in G has value (F1, F2).

---------------------
The optimal flow F1 has value 2.0
The optimal flow F2 has value 2.0
```

A similar syntax can be used for multiple sources-single sink flows.

Quadratic constraints

Quadratic inequalities are entered in the following way:

```
>>> t**2 > 2*t - alpha + x[1]*x[2]
<Quadratic Constraint: -t^2 + 2·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], Z[1][:,1]⟩ - (3·⟨[1], Y⟩ -␣
↪alpha) ≤ 0>
```

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

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 $[t; x]$ 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 $[t; u; x]$ inside a rotated version of the Lorretz cone.

A few examples:

```
>>> # A simple ice-cream cone constraint
>>> abs(x) < (2|x-1)
<5×1 SOC Constraint: ‖x‖ ≤ ⟨[2], x - [1]⟩>
>>> # SOC constraint with Frobenius norm
```

(continues on next page)

(continued from previous page)

```
>>> abs(Y+Z[0].T) < t+alpha
<9×1 SOC Constraint: ‖Y + Z[0]ᵀ‖ ≤ t + alpha>
>>> # Rotated SOC constraint
>>> abs(Z[1][:,0])**2 < (2*t-alpha)*(x[2]-x[-1])
<6×1 RSOC Constraint: ‖Z[1][:,0]‖² ≤ (2·t - alpha)·(x[2] - x[-1])>
>>> # t**2 is understood as the squared norm of [t]
>>> t**2 < D['Elisa']+t
<3×1 RSOC Constraint: ‖t‖² ≤ D[Elisa] + t>
>>> # 1 is understood as the squared norm of [1]
>>> 1 < (t-1)*(x[2]+x[3])
<3×1 RSOC Constraint: ‖1^(1/2)‖² ≤ (t - 1)·(x[2] + x[3])>
```

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)]) >> b[4]*b[4].T
<4×4 LMI Constraint: ∑(x[i]·b[i]·b[i]ᵀ : i ∈ [0...3]) ≽ b[4]·b[4]ᵀ>
```

Note the difference with

```
>>> pic.sum([x[i]*b[i]*b[i].T for i in range(4)]) > b[4]*b[4].T
<4×4 Affine Constraint: ∑(x[i]·b[i]·b[i]ᵀ : i ∈ [0...3]) ≥ 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')
>>> C = sdp.add_constraint(X >> 0)
>>> print(sdp)    #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (SDP):
10 variables, 0 affine constraints, 10 vars in 1 SD cones
<BLANKLINE>
X   : (4, 4), symmetric
<BLANKLINE>
    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.

Inequalities involving geometric means

It is possible to enter an inequality of the form

$$t \leq \prod_{i=1}^{n} x_i^{1/n}$$

in PICOS, where t is a scalar affine expression and x is an affine expression of dimension n (possibly a matrix, in which case x_i is counted in column major order). This inequality is internally converted to an equivalent set of second order cone inequalities, by using standard techniques (cf. e.g. *[1]*).

Many convex constraints can be formulated using inequalities that involve a geometric mean. For example, $t \leq x_1^{2/3}$ is equivalent to $t \leq t^{1/4} x_1^{1/4} x_1^{1/4}$, which can be entered in PICOS thanks to the function *picos.geomean()* :

```
>>> t < pic.geomean(t //x[1] //x[1] //1)
<Geometric Mean Constraint: t ≤ geomean([t; x[1]; x[1]; 1])>
```

Note that the latter example can also be passed to picos in a more simple way, thanks to an overloading of the `**` exponentiation operator:

```
>>> t < x[1]**(2./3)
<Power Constraint: x[1]^(2/3) ≥ t>
```

Inequalities involving geometric means are stored in a temporary object of the class *GeoMeanConstraint*, which can be passed to a problem with *add_constraint()*:

```
>>> geom_ineq = prob.add_constraint(t<pic.geomean(Y[:6]))
```

The constraint has an attribute `constraints` which contains auxiliary SOC inequalities that are used internally to represent the geometric mean:

```
>>> pprint(geom_ineq.constraints)
(<3×1 RSOC Constraint: ‖u[1:0-1]‖² ≤ Y[:6][0]·Y[:6][1]>,
 <3×1 RSOC Constraint: ‖u[1:2-3]‖² ≤ Y[:6][2]·Y[:6][3]>,
 <3×1 RSOC Constraint: ‖u[1:4-5]‖² ≤ Y[:6][4]·Y[:6][5]>,
 <3×1 RSOC Constraint: ‖u[2:0-3]‖² ≤ u[1:0-1]·u[1:2-3]>,
 <3×1 RSOC Constraint: ‖u[2:4-x]‖² ≤ u[1:4-5]·t>,
 <3×1 RSOC Constraint: ‖t‖² ≤ u[2:0-3]·u[2:4-x]>)
```

Likewise, auxiliary variables can be accessed via the `variables` attribute:

```
>>> pprint(geom_ineq.variables)
(<1×1 Continuous Variable: u[1:0-1]>,
 <1×1 Continuous Variable: u[1:2-3]>,
 <1×1 Continuous Variable: u[1:4-5]>,
 <1×1 Continuous Variable: u[2:0-3]>,
 <1×1 Continuous Variable: u[2:4-x]>)
```

Inequalities involving real powers or trace of matrix powers

As mentionned above, the `**` exponentiation operator has been overloaded to support real exponents. A rational approximation of the exponent is used, and the inequality are internally reformulated as a set of equivalent SOC inequalities. Note that only inequalities defining a convex regions can be passed:

```
>>> t**0.6666 > x[0]
<Power Constraint: t^(2/3) ≥ x[0]>
>>> t**-0.5 < x[0]
<Power Constraint: t^(-1/2) ≤ x[0]>
>>> try:
...     t**-0.5 > x[0]
```

(continues on next page)

(continued from previous page)

```
... except Exception as ex:
...     print('Exception: '+str(ex)) #doctest: +NORMALIZE_WHITESPACE
Exception: May only lower bound a concave power (p >= 0 and p <= 1).
>>> t**2 < x[1]+x[2]
<3×1 RSOC Constraint: ‖t‖^2 ≤ x[1] + x[2]>
```

More generally, inequalities involving trace of matrix powers can be passed to PICOS, by using the *picos.tracepow()* function. The following example creates the constraint

$$\text{trace } \left(x_0 A_0 A_0^T + x_2 A_2 A_2^T\right)^{2.5} \leq 3.$$

```
>>> pic.tracepow(x[0] * A[0]*A[0].T + x[2] * A[2]*A[2].T, 2.5) <= 3
<Trace of Power Constraint: trace((x[0]·A[0]·A[0]^T + x[2]·A[2]·A[2]^T)^(5/2)) ≤ 3>
```

Warning: when a power expression x^p (resp. the trace of matrix power $\text{trace } X^p$) is used, the base x is forced to be nonnegative (resp. the base X is forced to be positive semidefinite) by picos.

When the exponent is $0 < p < 1$, it is also possible to represent constraints of the form $\text{trace}(MX^p) \geq t$ with SDPs, where $M \succeq 0$, see *[2]*.

```
>>> pic.tracepow(X, 0.6666, coef = A[0].T*A[0]) >= t
<Trace of Power Constraint: trace(A[0]^T·A[0]·X^(2/3)) ≥ t>
```

As for geometric means, inequalities involving real powers yield their internal representation via the `constraints` and `variables` attributes.

Inequalities involving generalized p-norm

Inequalities of the form $\|x\|_p \leq t$ can be entered by using the function *picos.norm()*. This function is also defined for $p < 1$ by the usual formula $\|x\|_p := \left(\sum_i |x_i|^p\right)^{1/p}$. The norm function is convex over $\mathbb{R}^n$ for all $p \geq 1$, and concave over the set of vectors with nonnegative coordinates for $p \leq 1$.

```
>>> pic.norm(x,3) < t
<p-Norm Constraint: ‖x‖_3 ≤ t>
>>> pic.norm(x,'inf') < 2
<p-Norm Constraint: ‖x‖_inf ≤ 2>
>>> pic.norm(x,0.5) > x[0]-x[1]
<Generalized p-Norm Constraint: ‖x‖_(1/2) ≥ x[0] - x[1]>
```

Warning: Note that when a constraint of the form `norm(x,p) >= t` is entered (with $p \leq 1$), PICOS forces the vector `x` to be nonnegative (componentwise).

Inequalities involving the generalized $L_{p,q}$ norm of a matrix can also be handled with picos, cf. the documentation of *picos.norm()* .

As for geometric means, inequalities involving p-norms yield their internal representation via the `constraints` and `variables` attributes.

Inequalities involving the nth root of a determinant

The function *picos.detrootn()* can be used to enter the n-th root of the determinant of a $(n \times n)$-symmetric positive semidefinite matrix:

```
>>> M = sdp.add_variable('M',(5,5),'symmetric')
>>> t < pic.detrootn(M)
<n-th Root of a Determinant Constraint: det(M)^(1/5) ≥ t>
```

Warning: Note that when a constraint of the form `t < pic.detrootn(M)` is entered (with $p \leq 1$), PICOS forces the matrix `M` to be positive semidefinite.

As for geometric means, inequalities involving the nth root of a determinant yield their internal representation via the `constraints` and `variables` attributes.

2.1.8 Set membership

Since Picos 1.0.2, there is a *Set* class that can be used to pass constraints as membership of an affine expression to a set.

Following sets are currently supported:

- L_p- balls representing the set $\{x : \|x\|_p \leq r\}$ can be constructed with the function *pic.ball()*
- The standard simplex (scaled by a factor γ) $\{x \geq 0 : \sum_i x_i \leq r\}$ can be constructed with the function *pic.simplex()*
- Truncated simplexes $\{0 \leq x \leq 1 : \sum_i x_i \leq r\}$ and symmetrized Truncated simplexes $\{x : \|x\|_\infty \leq 1, \|x\|_1 \leq r\}$ can be constructed with the function *pic.truncated_simplex()*

Membership of an affine expression to a set can be expressed with the overloaded operator <<. This returns a temporary object that can be passed to a picos problem with the function *add_constraint()* .

```
>>> x << pic.simplex(1)
<Standard Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 1}>
>>> x << pic.truncated_simplex(2)
<Truncated Simplex Constraint: x ∈ {0 ≤ x ≤ 1 : ∑(x) ≤ 2}>
>>> x << pic.truncated_simplex(2,sym=True)
<Symmetrized Truncated Simplex Constraint: x ∈ {-1 ≤ x ≤ 1 : ∑(|x|) ≤ 2}>
>>> x << pic.ball(3)
<5×1 SOC Constraint: ‖x‖ ≤ 3>
>>> pic.ball(2,'inf') >> x
<p-Norm Constraint: ‖x‖_inf ≤ 2>
>>> x << pic.ball(4,1.5)
<p-Norm Constraint: ‖x‖_(3/2) ≤ 4>
```

2.1.9 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 LP to a **.lp** file:

```
import picos as pic
prob = pic.Problem(verbose = False)
y = prob.add_variable('y',1)
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')
```

(continues on next page)

(continued from previous page)

```
print()
#and display the content of the freshly created file:
print(open('helloworld.lp').read())
```

Generated output:

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

x   : (1, 1), continuous
y   : (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 x <= -1.5
in1 : -1 y+ 1 x <= -0.7
Bounds
y free
x free
Generals
Binaries
End
```

2.1.10 Solve a Problem

To solve a problem, you have to use the method *solve()* of the class *Problem*. Alternatively, the functions *maximize(obj)* and *minimize(obj)* can be used to specify the objective function and call the solver in a single statement. These method accept 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:

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

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) #or directly P.maximize(objective)

#display the problem and solve it
print(P)
print('type:   '+P.type)
print('status: '+P.status)
P.solve(verbose=0,solver='cvxopt')
print('status: '+P.status)
print()

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

print('The optimal value of this problem is:')
print(P.obj_value()) # "print(objective)" would also work, because objective is␣
↪valued
print()

#--------------------#
#  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

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

c1=P.get_constraint(1)
print('The dual of the constraint {} is:'.format(c1))
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 ≤ [2×1]
---------------------
type:   LP
```

(continues on next page)

(continued from previous page)

```
status: unsolved
status: optimal

The optimal value of this problem is:
3.0...

The solution of the problem is:
[ 2.00e+00]
[ 2.00e+00]

The dual of the constraint x[0] ≥ x[1] is:
0.25...

And its slack is:
1.8...

The dual of the constraint A·x ≤ [2×1] is:
[ 4.56e-10]
[ 7.50e-01]

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

2.1.11 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 $\|\mathbf{z}\| \leq \lambda$.

For rotated second order cone constraints of the form $\|\mathbf{x}\|^2 \leq tu$ where $\mathbf{x}$ is a vector of dimension n, the dual variable is a vector of dimension $n+2$ of the form $[\alpha, \beta, \mathbf{z}]$ with $\alpha, \beta \geq 0$, where the n-dimensional vector $\mathbf{z}$ satisfies $\|\mathbf{z}\|^2 \leq 4\alpha\beta$.

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

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

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 ($M_k \in \mathbb{S}_m$).

Its dual problem can be written as:

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

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 $\mu_j^r = [\alpha_j; \beta_j; \mathbf{z_j^r}]$ for the constraint

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

- The symmetric positive definite matrix X for the constraint

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

2.1.12 References

1. "*Applications of second-order cone programming*", M.S. Lobo, L. Vandenberghe, S. Boyd and H. Lebret, *Linear Algebra and its Applications*, 284, p. *193-228*, 1998.
2. "On the semidefinite representations of real functions applied to symmetric matrices", G. Sagnol, *Linear Algebra and its Applications*, 439(10), p. *2829-2843*, 2013.

2.2 Examples

2.2.1 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 a graph generated by the LCF generator of the networkx package. The graph and the edge capacities are deterministic, so that you can compare your results.

```
import picos as pic
import networkx as nx
import pylab
import random

# Use a fixed RNG seed so the result is reproducable.
random.seed(1)
```

(continues on next page)

(continued from previous page)

```
# Number of nodes.
N=20

# Generate a graph using LCF notation.
G=nx.LCF_graph(N,[1,3,14],5)
G=nx.DiGraph(G) #edges are bidirected

# Generate edge capacities.
c={}
for e in sorted(G.edges(data=True)):
  capacity = random.randint(1, 20)
  e[2]['capacity'] = capacity
  c[(e[0], e[1])]  = capacity

# Convert the capacities to a PICOS expression.
cc=pic.new_param('c',c)

# Manually set a layout for which the graph is planar.
pos={
  0:  (0.07, 0.70), 1:  (0.18, 0.78), 2:  (0.26, 0.45), 3:  (0.27, 0.66),
  4:  (0.42, 0.79), 5:  (0.56, 0.95), 6:  (0.60, 0.80), 7:  (0.64, 0.65),
  8:  (0.55, 0.37), 9:  (0.65, 0.30), 10: (0.77, 0.46), 11: (0.83, 0.66),
  12: (0.90, 0.41), 13: (0.70, 0.10), 14: (0.56, 0.16), 15: (0.40, 0.17),
  16: (0.28, 0.05), 17: (0.03, 0.38), 18: (0.01, 0.66), 19: (0.00, 0.95)
}

# Set source and sink nodes for flow computation.
s=16
t=10

# Set node colors.
node_colors=['lightgrey']*N
node_colors[s]='lightgreen' # Source is green.
node_colors[t]='lightblue'  # Sink is blue.

# Define a plotting helper that closes the old and opens a new figure.
def new_figure():
  try:
    global fig
    pylab.close(fig)
  except NameError:
    pass
  fig=pylab.figure(figsize=(11,8))
  fig.gca().axes.get_xaxis().set_ticks([])
  fig.gca().axes.get_yaxis().set_ticks([])

# Plot the graph with the edge capacities.
new_figure()
nx.draw_networkx(G, pos, node_color=node_colors)
labels={
  e: '{} | {}'.format(c[(e[0], e[1])], c[(e[1], e[0])])
  for e in G.edges if e[0] < e[1]}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
pylab.show()
```

The first number on an edge label denotes the capacity from the node with the smaller number to the node with the larger number; the second number denotes the capacity for the other direction. Source and sink that we will use for flow computations are drawn in green and blue, respectively.

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, $\forall e \in E,\ f(e) \leq c(e)$, and
- the flow is conserved at each non-terminal node, $\forall n \in V \backslash \{s,t\},\ \sum_{(i,n)\in E} f((i,n)) = \sum_{(n,j)\in E} f((n,j))$.

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

```
maxflow=pic.Problem()

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

# Add another variable for the total flow.
F=maxflow.add_variable('F',1)

# Enforce edge capacities.
maxflow.add_list_of_constraints(
  [f[e]<cc[e] for e in G.edges()], # list of constraints
  [('e',2)],                      # e is a double index
  'edges')                        # set the index belongs to

# Enforce 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)')

# Set 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)'))

# Set 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)

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

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

# Solve the problem.
maxflow.solve(verbose=0,solver='glpk')
```

An equivalent and faster way to define this problem is to use the function *flow_Constraint*:

```
maxflow2=pic.Problem()

# Add the flow variables.
f={}
for e in G.edges():
  f[e]=maxflow2.add_variable('f[{0}]'.format(e),1)

# Add another variable for the total flow.
F=maxflow2.add_variable('F',1)

# Enforce all flow constraints at once.
maxflow2.add_constraint(pic.flow_Constraint(
  G, f, source=16, sink=10, capacity='capacity', flow_value=F, graphName='G'))

# Set the objective.
maxflow2.set_objective('max',F)

# Solve the problem.
maxflow2.solve(verbose=0,solver='glpk')
```

Let us now draw the maximum flow computed with the second approach:

```
# Close the old figure and open a new one.
new_figure()

# Determine which edges carry flow.
flow_edges=[e for e in G.edges() if f[e].value > 1e-4]

# Draw the nodes and the edges that don't carry flow.
nx.draw_networkx(G, pos, edge_color='lightgrey', node_color=node_colors,
  edgelist=[e for e in G.edges
    if e not in flow_edges and (e[1], e[0]) not in flow_edges])

# Draw the edges that carry flow.
nx.draw_networkx_edges(G, pos, edgelist=flow_edges)

# Show flow values and capacities on these edges.
labels={e: '{0}/{1}'.format(f[e], c[e]) for e in flow_edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Show the maximum flow value.
fig.suptitle("Maximum flow value: {}".format(F), fontsize=16, y=0.95)

# Show the figure.
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 that carries flow.

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:

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

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 min-cut problem below, again for `s=16` and `t=10`:

```
mincut=pic.Problem()

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

# Add variables for the potentials.
p=mincut.add_variable('p',N)

# State the 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

# Set the source potential to one.
mincut.add_constraint(p[s]==1)

# Set the sink potential to zero.
mincut.add_constraint(p[t]==0)

# Enforce 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
        'edges')                     # set the index belongs to

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

mincut.solve(verbose=0,solver='glpk')

# Determine the cut edges and node sets.
# Rounding is done because solvers might return near-optimal solutions due to
# numerical precision issues.
cut=[e for e in G.edges() if abs(d[e].value-1) < 1e-6]
S  =[n for n in G.nodes() if abs(p[n].value-1) < 1e-6]
T  =[n for n in G.nodes() if abs(p[n].value  ) < 1e-6]
```

Note that the minimum-cut can also be derived from the dual variables of the max-flow LP:

Note: Due to a technical issue, the interactive Python listing below is not automatically verified. If you find an error, please refer to *the section on reporting a bug*.

```
>>> # capacited flow constraint
>>> capaflow=maxflow.get_constraint((0,))
>>> dualcut=[e for i,e in enumerate(G.edges()) if abs(capaflow[i].dual[0]-1)<1e-6]
>>> # 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 abs(consflow[i].dual[0]-1)<1e-6]
>>> Tdual = [t]+ [n for i,n in
...           enumerate([n for n in G.nodes() if n not in (s,t)])
...           if abs(consflow[i].dual[0])<1e-6]
>>> cut == dualcut
True
>>> set(S) == set(Sdual)
True
>>> set(T) == set(Tdual)
True
```

Let us now draw the minim cut:

```
# Close the old figure and open a new one.
new_figure()

# Draw the nodes and the edges that are not in the cut.
nx.draw_networkx(G, pos, node_color=node_colors,
  edgelist=[e for e in G.edges() if e not in cut and (e[1], e[0]) not in cut])

# Draw edges that are in the cut.
nx.draw_networkx_edges(G, pos, edgelist=cut, edge_color='r')

# Show capacities for cut edges.
labels={e: '{}'.format(c[e]) for e in cut}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_color='r')

# Show the minimum cut value and the partition.
fig.suptitle("Minimum cut value: {}\nS: {}, T: {}".format(
  mincut.obj_value(), S, T), fontsize=16, y=0.97)

# Show the figure.
pylab.show()
```

The graph shows the source in blue, the sink in green, and the edges defining the cut in red, with their capacities.

Multicut (MIP)

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

We can obtain a MIP formulation of the multicut problem via a small modification of the min-cut LP. The idea is to introduce a different potential for every node that is the source of a pair in $\mathcal{P}$, that is

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

and to constrain the cut indicator variables 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 min-cut 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()

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

# Extract the sources and sinks.
sources=set([p[0] for p in pairs])
sinks=set([p[1] for p in pairs])

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

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

# State the 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()],
  ['i','j','s'], 'edges x sources')

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

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

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

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

# Solve the problem.
multicut.solve(verbose=0,solver='glpk')

# Extract the cut.
cut=[e for e in G.edges() if y[e].value==1]
```

Let us now draw the multicut:

```
# Close the old figure and open a new one.
new_figure()

# Define matching colors for the pairs.
colors=[
  ('#4CF3CE','#0FDDAF'), # turquoise
  ('#FF4D4D','#FF0000'), # red
  ('#FFA64D','#FF8000'), # orange
  ('#3ABEFE','#0198E1'), # topaz
  ('#FFDB58','#FFCC11'), # mustard
  ('#BCBC8F','#9F9F5F')  # khaki
]

# Assign the colors.
node_colors=['lightgrey']*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]

# Draw the nodes and the edges that are not in the cut.
nx.draw_networkx(G, pos, node_color=node_colors,
  edgelist=[e for e in G.edges() if e not in cut and (e[1], e[0]) not in cut])

# Draw the edges that are in the cut.
nx.draw_networkx_edges(G, pos, edgelist=cut, edge_color='r')

# Show capacities for cut edges.
labels={e: '{}'.format(c[e]) for e in cut}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_color='r')

# Show the cut capacity.
fig.suptitle("Multicut value: {}"
  .format(multicut.obj_value()), fontsize=16, y=0.95)

# Show the figure.
pylab.show()
```

The graph shows terminal nodes with matching hue. Sources are a tad lighter than sinks to make them distinguishable. The edges defining the cut are drawn in red and show their capacities. The colors for the source nodes are, in order: Turquoise, red, orange, topaz, mustard and khaki.

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 whether $n \in S$ 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 and of rank 1, we obtain an SDP by relaxing the rank-one constraint on X:

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

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. We give a simple implementation of their

algorithm. First, let us define and solve the SDP relaxation:

```
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()

# Add the symmetric matrix variable.
X=maxcut.add_variable('X',(N,N),'symmetric')

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

# Constrain X to have ones on the diagonal.
maxcut.add_constraint(pic.tools.diag_vect(X)==1)

# Constrain X to be positive semidefinite.
maxcut.add_constraint(X>>0)

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

#print(maxcut)

# Solve the problem.
maxcut.solve(verbose = 0,solver='cvxopt')

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

Then, we perform the random projection:

```
# Use a fixed RNG seed so the result is reproducable.
cvx.setseed(1)

# Perform a 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

# Do up to 100 projections. Stop if we are within a factor 0.878 of the SDP
# optimal value.
count=0
obj_sdp=maxcut.obj_value()
obj=0
while (count < 100 or obj < 0.878*obj_sdp):
  r=cvx.normal(20,1)
  x=cvx.matrix(np.sign(V*r))
  o=(x.T*L*x).value
  if o > obj:
    x_cut=x
    obj=o
  count+=1
```

(continues on next page)

(continued from previous page)

```
x=x_cut

# Extract the cut and the seperated node sets.
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]
leave = [e for e in G.edges if e not in cut]
```

Let us now draw this cut:

```
# Close the old figure and open a new one.
new_figure()

# Assign colors based on set membership.
node_colors=[('lightgreen' if n in S1 else 'lightblue') for n in range(N)]

# Draw the nodes and the edges that are not in the cut.
nx.draw_networkx(G, pos, node_color=node_colors, edgelist=leave)
labels={e: '{}'.format(G[e[0]][e[1]]['weight']) for e in leave}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Draw the edges that are in the cut.
nx.draw_networkx_edges(G, pos, edgelist=cut, edge_color='r')
labels={e: '{}'.format(G[e[0]][e[1]]['weight']) for e in cut}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_color='r')

# Show the relaxation optimum value and the cut capacity.
rval = maxcut.obj_value()
sval = sum(G[e[0]][e[1]]['weight'] for e in cut)
fig.suptitle(
  'SDP relaxation value: {0:.1f}\nCut value: {1:.1f} = {2:.3f}×{0:.1f}'
  .format(rval, sval, sval/rval), fontsize=16, y=0.97)

# Show the figure.
pylab.show()
```

The graph shows the edges defining the cut in red. The nodes are colored blue or green depending on the partition that they belong to.

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.

2.2.2 Complex Semidefinite Programming

Since the version 1.0.1, it is possible to do complex semidefinite programming with Picos. This extension of semidefinite programming to the complex domain was introduced by Goemans and Williamson *[1]* as relaxtions of combinatorial optimization problems, and has applications e.g. in Quantum Information Theory *[2]*, or for the phase recovery problem in signal processing *[3]*.

To handle complex SDPs in Picos, we have introduced two new variable types: `'complex'` and `'hermitian'`. A complex variable can be created as follows:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> P = pic.Problem()
>>> Z = P.add_variable('Z',(3,2),'complex')
```

it automatically creates two variables called `Z_RE` and `Z_IM` which contains the real and imaginary part of `Z`, and that be accessed by using the `real` and `imag` properties:

```
>>> Z.real
<3×2 Continuous Variable: Z_RE>
>>> Z.imag
<3×2 Continuous Variable: Z_IM>
>>> Z.vtype
'complex'
```

The python variable `Z` contains an affine expression equal to `Z_RE + 1j * Z_IM`, and that can be used to easily define a complex SDP.

The variable type `'hermitian'` can be used to create a complex variable that is forced to be Hermitian. The following properties can now be used with every affine expression: `conj` (complex conjugate), `real` (real part, i.e. `exp.real` returns `0.5 * (exp+exp.conj)`), `imag` (imaginary part, i.e. `exp.imag` returns `-0.5 * 1j * (exp-exp.conj)`), anf `H` (Hermitian transposition, i.e. `exp.H` returns `exp.conj.T`).

```
>>> X = P.add_variable('X',(3,3),'hermitian')
>>> X >> 0
<3×3 LMI Constraint: X ≽ 0>
```

Fidelity in Quantum Information Theory

The material of this section is inspired from a lecture of John Watrous *[4]*.

The Fidelity between two (Hermitian) positive semidefinite operators P and Q is defined as:

$$F(P,Q) = \|P^{1/2}Q^{1/2}\|_{tr} = \max_U \left|\text{trace}\left(P^{1/2}UQ^{1/2}\right)\right|,$$

where the trace norm $\|\cdot\|_{tr}$ is the sum of the singular values, and the maximization goes over the set of all unitary matrices U. This quantity can be expressed as the optimal value of the following complex-valued SDP:

$$\begin{aligned} \underset{Z\in\mathbb{C}^{n\times n}}{\text{maximize}} \quad & \frac{1}{2}\text{trace}(Z+Z^*) \\ \text{subject to} \quad & \begin{pmatrix} P & Z \\ Z^* & Q \end{pmatrix} \succeq 0 \end{aligned}$$

This Problem can be solved as follows in PICOS

```
#generate two (arbitrary) positive hermitian operators
P = cvx.matrix([ [1-1j , 2+2j  , 1    ],
                 [3j   , -2j   , -1-1j],
                 [1+2j, -0.5+1j, 1.5  ]
               ])
P = P * P.H

Q = cvx.matrix([ [-1-2j , 2j   , 1.5   ],
                 [1+2j  ,-2j   , 2.-3j ],
                 [1+2j  ,-1+1j , 1+4j  ]
               ])
Q = Q * Q.H

n=P.size[0]
```

(continues on next page)

(continued from previous page)

```
P = pic.new_param('P',P)
Q = pic.new_param('Q',Q)

#create the problem in picos

F = pic.Problem()
Z = F.add_variable('Z',(n,n),'complex')

F.set_objective('max','I'|0.5*(Z+Z.H))        #('I' | Z.real) works as well
F.add_constraint(((P & Z) // (Z.H & Q))>>0 )

print(F)

F.solve(verbose = 0, solver = 'cvxopt')

print('fidelity: F(P,Q) = {0:.4f}'.format(F.obj_value()))

print('optimal matrix Z:')
print(Z)

#verify that we get the same value with numpy
import numpy as np
PP = np.matrix(P.value)
QQ = np.matrix(Q.value)

S,U = np.linalg.eig(PP)
sqP = U * np.diag([s**0.5 for s in S]) * U.H #square root of P
S,U = np.linalg.eig(QQ)
sqQ = U * np.diag([s**0.5 for s in S]) * U.H #square root of P

fidelity = sum(np.linalg.svd(sqP * sqQ)[1])  #trace-norm of P**0.5 *  Q**0.5

print('fidelity computed by trace-norm: F(P,Q) = {0:.4f}'.format(fidelity))
```

```
---------------------
optimization problem (Complex SDP):
18 variables, 0 affine constraints, 21 vars in 1 SD cones

Z_IM        : (3, 3), continuous
Z_RE        : (3, 3), continuous

        maximize trace(0.5·(Z + Z^H))
such that
[P, Z; Z^H, Q] ⪰ 0
---------------------
fidelity: F(P,Q) = 37.4742
optimal matrix Z:
[ 1.51e+01+j2.21e+00 -7.17e+00-j1.22e+00  2.52e+00+j6.87e-01]
[-4.88e+00+j4.06e+00  1.00e+01-j1.57e-01  8.33e+00+j1.13e+01]
[-4.33e-01+j2.98e-01  3.84e+00-j3.28e+00  1.24e+01-j2.05e+00]

fidelity computed by trace-norm: F(P,Q) = 37.4742
```

Phase Recovery in Signal Processing

The material from this section is inspired from *[3]*.

The goal of the phase recovery problem is to reconstruct the complex phase of a vector, when we are only given the magnitude of some linear measurements. This problem can be formulated as a nonconvex optimization problem, and the authors of *[3]* have proposed a complex SDP relaxation similar to the well known *Max-Cut* SDP: Given a linear operator A and a vector b of measured amplitudes, define the positive semidefinite hermitian matrix

$M = \operatorname{diag}(b)(I - AA^{\dagger})\operatorname{diag}(b)$, the *Phase-cut* Problem is:

$$\begin{array}{ll} \underset{U \in \mathbb{H}_n}{\text{minimize}} & \langle U, M \rangle \\ \text{subject to} & \operatorname{diag}(U) = 1 \\ & U \succeq 0 \end{array}$$

Here the variable U must be hermitian ($U \in \mathbb{H}_n$), and we have a solution to the phase recovery problem if $U = uu^*$ has rank one. Otherwise, the leading singular vector of U is used as an approximation.

This problem can be implemented as follows using Picos:

```
# We generate an arbitrary matrix M

import cvxopt as cvx
import picos as pic

n    = 5
rank = 4  #we take a singular M for the sake of generality

M = cvx.normal (n,rank) +1j*cvx.normal (n,rank)
M = M * M.H
M = pic.new_param('M',M)

P = pic.Problem()
U = P.add_variable('U',(n,n),'hermitian')
P.add_list_of_constraints([U[i,i]==1 for i in range(n)],'i')
P.add_constraint(U >> 0)

P.set_objective('min', U | M)

print(P)

#solve the problem
P.solve(verbose=0, solver='cvxopt')

#optimal complex variable
print()
print('optimal variable: U=')
print(U)
print()

#Do we have a matrix of rank one ?
S, V = np.linalg.eig(U.value)
print('rank of U = ', len([s for s in S if abs(s)>1e-6]))
```

```
---------------------
optimization problem  (SDP):
36 variables, 8 affine constraints, 36 vars in 1 SD cones

U       : (8, 8), hermitian

        minimize ⟨U, M⟩
such that
U[i,i] = 1 ∀ i ∈ [0...4]
U ≽ 0
---------------------

optimal variable: U=
[ 1.00e+00+j8.97e-10  9.11e-01-j1.27e-01  1.46e-01+j9.38e-01 -7.30e-01+j6.32e-01 -
↪5.52e-01-j8.09e-01]
```

(continues on next page)

(continued from previous page)

```
[ 9.11e-01+j1.27e-01  1.00e+00+j1.32e-09  1.05e-01+j9.56e-01 -8.04e-01+j5.66e-01 -
↪4.73e-01-j8.40e-01]
[ 1.46e-01-j9.38e-01  1.05e-01-j9.56e-01  1.00e+00+j1.19e-09  4.96e-01+j8.58e-01 -
↪9.00e-01+j4.19e-01]
[-7.30e-01-j6.32e-01 -8.04e-01-j5.66e-01  4.96e-01-j8.58e-01  1.00e+00+j9.45e-10 -
↪9.85e-02+j9.91e-01]
[-5.52e-01+j8.09e-01 -4.73e-01+j8.40e-01 -9.00e-01-j4.19e-01 -9.85e-02-j9.91e-01 ␣
↪1.00e+00+j9.16e-10]


rank of U =  2
```

References

1. "Approximation algorithms for MAX-3-CUT and other problems via complex semidefinite programming", M.X. Goemans and D. Williamson. In Proceedings of the thirty-third annual *ACM symposium on Theory of computing*, pp. 443-452. ACM, 2001.
2. "Semidefinite programs for completely bounded norms", J. Watrous, arXiv preprint 0901.4709, 2009.
3. "Phase recovery, maxcut and complex semidefinite programming", I. Waldspurger, A. d'Aspremont, and S. Mallat. *Mathematical Programming*, pp. 1-35, 2012.
4. "Semidefinite Programs for fidelity and optimal measurements", J. Watrous, in the script of a course on Theory of Quantum Information

2.2.3 Examples from Optimal Experimental Design

Optimal experimental design is a theory at the interface of statistics and optimization, which studies how to allocate some statistical trials within a set of available design points. 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: a trial in the i^{th} design point 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 error vectors ϵ_i are i.i.d. with a unit variance.

Several optimization criteria 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]]),
```

(continues on next page)

(continued from previous page)

```
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])
```

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 design point #5, and 87.2% on the design point #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)
```

(continues on next page)

(continued from previous page)

```
#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 design is:')
print(w)
```

Generated output:

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

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

        minimize ⟨[1], mu⟩
such that
  ‖z[i]‖ ≤ mu[i] ∀ i ∈ [0...7]
  ∑(A[i]·z[i] : i ∈ [0...7]) = c
---------------------

The optimal design 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
```

(continues on next page)

(continued from previous page)

```
        '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,solver='cvxopt')

mu = [cons.dual[0] for cons in prob_dual_c.get_constraint((0,))] #Lagrangian duals␣
↪of the SOC constraints
mu = cvx.matrix(mu)
w=mu/sum(mu) #normalize mu to get the optimal weights
print()
print('The optimal design 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 ∀ i ∈ [0...7]
---------------------

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

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 11 available design points, corresponding to the columns of the matrices `A[4]`, `A[5]`, `A[6]`, and `A[7][:,:-1]` defined in the preambule.

The optimal design allocates 3.37% to point #5 (2nd column of `A[5]`), 27.9% to point #7 (1st column of `A[6]`), 11.8% to point #8 (2nd column of `A[6]`), 27.6% to point #9 (3rd column of `A[6]`), and 29.3% to point #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
AA=AA[:-1] #remove the last design point (it is the same as the last-but-one)
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)
```

(continues on next page)

(continued from previous page)

```
#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,solver='cvxopt')

mu=mu.value
w=mu/sum(mu) #normalize mu to get the optimal weights
print()
print('The optimal design 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):
22 variables, 27 affine constraints

mu  : (11, 1), continuous
z   : list of 11 variables, (1, 1), continuous

        minimize ⟨[1], mu⟩
such that
  |z[i]| ≤ mu[i] ∀ i ∈ [0...10]
  ∑(A[i]·z[i] : i ∈ [0...10]) = c
---------------------

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

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 design point #5, and 87.2% on design point #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,solver='cvxopt')
w=mu.value
w=w/sum(w) #normalize mu to get the optimal weights
print()
print('The optimal design is:')
print(w)
```

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

mu  : (8, 1), continuous

        minimize ⟨[1], mu⟩
such that
∑(mu[i]·A[i]·A[i]^T : i ∈ [0...7]) ≽ c·c^T
mu ≥ 0
---------------------

The optimal design 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:

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

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

```
---------------------
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 ∀ i ∈ [0...7]
  X ⪰ 0
---------------------

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

(continues on next page)

(continued from previous page)

```
[ 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]
```

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 design point #3, 14.2% on point #4, 8.51% on point #5, 12.1% on point #6, 13.2% on point #7, and 27.0% on point #8.

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

```
---------------------
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] ∀ i ∈ [0...7]
  ∑(A[i]·Z[i] : i ∈ [0...7]) = I
---------------------

The optimal design 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,solver='cvxopt')

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

```
---------------------
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 ∀ i ∈ [0...7]
---------------------

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

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:

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

The optimal solution allocates 29.7% and 20.3% to the design points #3 and #4, and respectively 6.54%, 11.9%, 9.02% and 22.5% to the design points #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(
```

(continues on next page)

(continued from previous page)

```
        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,solver='cvxopt')
w=w.value
w=w/sum(w) #normalize w to get the optimal weights
print()
print('The optimal design is:')
print(w)
```

```
---------------------
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
  ∑(A[i]·Z[i] : i ∈ [0...7]) = I
  ⟨[1], w[:4]⟩ ≤ 0.5
  ⟨[1], w[4:]⟩ ≤ 0.5
  ‖Z[i]‖² ≤ mu[i]·w[i] ∀ i ∈ [0...7]
---------------------

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

Exact A-optimal design: MISOCP

In the exact version of A-optimality, a number $N \in \mathbb{N}$ of trials is given, and the goal is to find the optimal number of times $n_i \in \mathbb{N}$ that a trial on design point #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 trials 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(verbose = 0)
print(n)
```

```
---------------------
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] ∀ i ∈ [0...7]
```

(continues on next page)

(continued from previous page)

```
  ∑(A[i]·Z[i] : i ∈ [0...7]) = I
  ⟨[1], n⟩ ≤ N
---------------------
[...]
[...]
[ 5.00e+00]
[ 3.00e+00]
[ 2.00e+00]
[ 2.00e+00]
[ 3.00e+00]
[ 5.00e+00]
```

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

The D-optimal design problem has a SOCP formulation involving a geometric mean in the objective function:

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

By introducing a new variable t such that $t \leq \left(\prod_{i=1}^{m} L_{i,i}\right)^{1/m}$, we can pass this problem to PICOS with the function *picos.geomean()*, which reformulates the geometric mean inequality as a set of equivalent second order cone constraints. The example below allocates respectively 22.7%, 3.38%, 1.65%, 5.44%, 31.8% and 35.1% to the design points #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 variable to handle the geometric mean in the objective function
t= prob_D.add_variable('t',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]
```

(continues on next page)

(continued from previous page)

```
                                for i in range(s)],'i','[s]')
prob_D.add_constraint(1|w<1)
prob_D.add_constraint(t<pic.geomean(pic.diag_vect(L)))
prob_D.set_objective('max',t)

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

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

L   : (5, 5), continuous
V   : list of 8 variables, (3, 5), continuous
t   : (1, 1), continuous
w   : (8, 1), continuous

        maximize t
such that
  L = ∑(A[i]·V[i] : i ∈ [0...7])
  L[i,j] = 0 ∀ (i,j) ∈ zip([0,0,...,2,3],[1,2,...,4,4])
  ‖V[i]‖ ≤ m^0.5·w[i] ∀ i ∈ [0...7]
  ⟨[1], w⟩ ≤ 1
  t ≤ geomean(diag(L))
---------------------
[...]
[...]
[ 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 variable to handle the geomean inequality
t = prob_exact_D.add_variable('t',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')
```

(continues on next page)

(continued from previous page)

```
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)
prob_exact_D.add_constraint(t<pic.geomean( pic.diag_vect(L)))

prob_exact_D.set_objective('max',t)

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

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

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

        maximize t
such that
  L = ∑(A[i]·V[i] : i ∈ [0...7])
  L[i,j] = 0 ∀ (i,j) ∈ zip([0,0,...,2,3],[1,2,...,4,4])
  ‖V[i][:,j]‖^2 ≤ (n[i]/N)·T[i,k] ∀ (i,j) ∈ ???
  ⟨[1], T[:,i]⟩ ≤ 1 ∀ i ∈ ???
  ⟨[1], n⟩ ≤ N
  t ≤ geomean(diag(L))
---------------------
[...]
[...]
[ 5.00e+00]
[ 1.00e+00]
[...]
[ 1.00e+00]
[ 6.00e+00]
[ 7.00e+00]
```

Former MAXDET formulation of the D-optimal design: SDP

A so-called MAXDET Programming formulation of the D-optimal design has been known since the late 90's *[3]*, and can be reformulated as a SDP thanks to the `detrootn()` function. The following code finds the same design as the SOCP approach presented above.

```
#problem, variables and parameters
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)
w = prob_D.add_variable('w',s,lower=0)
t = prob_D.add_variable('t',1)
```

(continues on next page)

(continued from previous page)

```
#constraint and objective
prob_D.add_constraint(1|w < 1)
Mw = pic.sum([w[i]*AA[i]*AA[i].T for i in range(s)],'i')
prob_D.add_constraint(t < pic.detrootn(Mw))
prob_D.set_objective('max',t)

#solve and display
print(prob_D)
prob_D.solve(verbose=0,solver='cvxopt')
print(w)
```

```
---------------------
optimization problem (Conic Program):
29 variables, 1 affine constraints, 18 vars in 6 SO cones, 55 vars in 1 SD cones

t    : (1, 1), continuous
w    : (8, 1), continuous, nonnegative

        maximize t
such that
  ⟨[1], w⟩ ≤ 1
  det(∑(w[i]·A[i]·A[i]^T : i ∈ [0...7]))^(1/5) ≥ t
---------------------
[ ...]
[ ...]
[ 2.27e-01]
[ 3.38e-02]
[ 1.65e-02]
[ 5.44e-02]
[ 3.18e-01]
[ 3.51e-01]
```

General Phi_p optimal design Problem: SDP

The A- and D-optimal design problems presented above can be obtained as special cases of the general Kiefer Φ_p- optimal design problem, where p is a real in $(-\infty, 1]$:

$$\begin{aligned} \underset{w\in\mathbb{R}^s}{\text{maximize}} \quad & \left(\frac{1}{m}\,\text{trace}\,(\sum_{i=1}^{s} w_i A_i A_i^T)^p\right)^{1/p} \\ \text{subject to} \quad & w \geq 0,\ \sum_{i=1}^{s} w_i \leq 1. \end{aligned}$$

These problems are easy to enter in PICOS, thanks to the *tracepow()* function, that automatically replaces inequalities involving trace of matrix powers as a set of equivalent linear matrix inequalities (SDP) (cf. *[4]*). Below are two examples with $p = 0.2$ and $p = -3$, allocating respectively (20.6%, 0.0%, 0.0%, 0.92%, 40.8%, 37.7%), and (24.8%, 16.6%, 10.8%, 14.1%, 7.84%, 26.0%) of the trials to the design points 3 to 8.

```
#problems, variables and parameters
prob_0dot2  = pic.Problem()
probminus3 = 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)

w02 = prob_0dot2.add_variable('w',s,lower=0)
```

(continues on next page)

(continued from previous page)

```
wm3 = probminus3.add_variable('w',s,lower=0)

t02 = prob_0dot2.add_variable('t',1)
tm3 = probminus3.add_variable('t',1)


#constraint and objective
prob_0dot2.add_constraint(1|w02 < 1)
probminus3.add_constraint(1|wm3 < 1)

Mw02 = pic.sum([w02[i]*AA[i]*AA[i].T for i in range(s)],'i')
prob_0dot2.add_constraint(t02 < pic.tracepow(Mw02,0.2))
prob_0dot2.set_objective('max',t02)

Mwm3 = pic.sum([wm3[i]*AA[i]*AA[i].T for i in range(s)],'i')
probminus3.add_constraint(tm3 > pic.tracepow(Mwm3,-3))
probminus3.set_objective('min',tm3)

#solve and display
prob_0dot2.solve(verbose=0,solver='cvxopt')
probminus3.solve(verbose=0,solver='cvxopt')

print('*** p=0.2 ***')
print(prob_0dot2)
print(w02)

print('*** p= -3 ***')
print(probminus3)
print(wm3)
```

```
*** p=0.2 ***
---------------------
optimization problem (SDP):
54 variables, 2 affine constraints, 165 vars in 3 SD cones

t    : (1, 1), continuous
w    : (8, 1), continuous, nonnegative

        maximize t
such that
  ⟨[1], w⟩ ≤ 1
  trace(∑ (w[i]·A[i]·A[i]^T : i ∈ [0...7])^(1/5)) ≥ t
---------------------
[ ...]
[ ...]
[ 2.06e-01]
[ ...]
[ ...]
[ 9.21e-03]
[ 4.08e-01]
[ 3.77e-01]

*** p= -3 ***
---------------------
optimization problem (SDP):
39 variables, 2 affine constraints, 110 vars in 2 SD cones

t    : (1, 1), continuous
w    : (8, 1), continuous, nonnegative

        minimize t
```

(continues on next page)

(continued from previous page)

```
such that
  ⟨[1], w⟩ ≤ 1
  trace(∑ (w[i]·A[i]·A[i]^T : i ∈ [0...7])^-3) ≤ t
--------------------
[ ...]
[ ...]
[ 2.48e-01]
[ 1.66e-01]
[ 1.08e-01]
[ 1.41e-01]
[ 7.83e-02]
[ 2.60e-01]
```

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. "Computing exact D-optimal designs by mixed integer second order cone programming", G. Sagnol and R. Harman, Submitted: arXiv:1307.4953.
3. "Determinant maximization with linear matrix inequality constraints", L. Vandenberghe, S. Boyd and S.P. Wu, *SIAM journal on matrix analysis and applications*, 19(2), 499-533, 1998.
4. "On the semidefinite representations of real functions applied to symmetric matrices", G. Sagnol, *Linear Algebra and its Applications*, 439(10), p. *2829-2843*, 2013.

2.3 Cheat Sheet

2.3.1 Manipulate expressions

Operator	Interpretation
`+`	addition
`+=`	inplace addition
`-`	substraction
`*`	multiplication
`^`	Hadamard (elementwise) product
`|`	scalar product
`/`	division
`**`	exponentiation
`abs()`	Euclidean (or Frobenius) norm
`[]`	slicing
`&`	horizontal concatenation
`//`	vertical concatenation
`.T`	transposition
`.H`	Hermitian transposition
`.Tx`	partial transposition
`.conj`	complex conjugate
`.real`	real part
`.imag`	imaginary part

2.3.2 Create constraints

Operator	Interpretation
< or <=	less or equal
> or >=	larger or equal
==	equal
<<	Löwner ordering $\preceq$, or set membership $\in$
>>	Löwner ordering $\succeq$, or set membership $\ni$

2.3.3 Create affine expressions

function	short doc
`sum()`	sums a list of affine expressions
`diag()`	diagonal matrix defined by its diagonal
`diag_vect()`	vector of diagonal elements of a matrix
`new_param()`	constant affine expression
`trace()`	trace of a square affine expression
`lowtri()`	vector of lower triangular elements
`partial_transpose()`	partial transposition
`partial_trace()`	partial trace

2.3.4 Create convex expressions

function	short doc
`geomean()`	geometric mean
`norm()`	(generalized) L_p- norm
`tracepow()`	trace of a p-th matrix power
`detrootn()`	n-th root of determinant
`sum_k_largest()`	sum of k largest elements
`sum_k_smallest()`	sum of k smallest elements
`sum_k_largest_lambda()`	sum of k largest eigenvalues
`sum_k_smallest_lambda()`	sum of k smallest eigenvalues
`lambda_max()`	largest eigenvalue
`lambda_min()`	smallest eigenvalue

2.3.5 Create sets

function	short doc
`ball(r,p)`	a L_p- ball of radius `r`
`simplex(a)`	a standard simplex $\{x \geq 0 : \|x\|_1 \leq a\}$
`truncated_simplex(a)`	a set of the form $\{0 \leq x \leq 1 : \|x\|_1 \leq a\}$, or $\{x : \|x\|_\infty \leq 1; \|x\|_1 \leq a\}$

2.3.6 Other useful functions

Transform a problem

function	short doc
`convert_quad_to_socp()`	replaces quadratic constraints by equivalent second order cone constraints
`as_real()`	transform complex SDP to equivalent real-valued SDP
`as_dual()`	returns Lagrangian dual of a problem

Get information on a problem

function	short doc
get_variable(name)	gets the variable object `name`
get_valued_variable(name)	gets the value of the variable `name`
check_current_value_feasibility()	are the current variable value feasible?
obj_value()	objective for the current variable values
.type	returns problem's type

Miscellaneous

function	short doc
available_solvers()	lists installed solvers
import_cbf()	imports data from a .cbf file
eval_dict()	evaluates a dictionary of picos variables (after a problem has been solved)
write_to_file()	writes problem to a file

2.4 API Reference

PICOS is organized in a number of submodules, most of which you do not need to access directly when solving optimization problems. It is usually sufficient to `import picos` and use the functions and classes provided in the *picos* namespace. Additional utilities can be found in *picos.tools*.

2.4.1 picos

The *picos* namespace gives you quick access to the most important classes and functions for optimizing with PICOS, so that `import picos` is often sufficient for implementing your model. You can find additional utilities in the *picos.tools* namespace.

Functions

ascii()	Let PICOS create future string representations using only ASCII characters.
ball(r[, p])	returns a *Ball* object representing:
default_charset([rebuildDerivedGlyphs])	Let PICOS create future string representations using unicode characters.
detrootn(exp)	returns a *DetRootN_Exp* object representing the determinant of the n th-root of the symmetric matrix `exp`, where n is the dimension of the matrix.
diag(exp[, dim])	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.
diag_vect(exp)	Returns the vector with the diagonal elements of the matrix expression `exp`
exp(x)	Exponentiation of a PICOS, CVXOPT, NumPy, or numeric expression.
expcone()	returns a *ExponentialCone* object representing the set closure $\{(x, y, z) : y > 0, y \exp(x/y) \leq z\}$
flow_Constraint(G, f, source, sink, flow_value)	Constructs a network flow constraint.
geomean(exp)	returns a *GeoMeanExp* object representing the geometric mean of the entries of `exp[:]`.
get_version_info()	

Continued on next page

Table 1 – continued from previous page

import_cbf(filename)	Imports the data from a CBF file, and creates a *Problem* object.
kron(A, B)	Kronecker product of 2 expression, at least one of which must be constant
kullback_leibler(x[, y])	A shorthand for *KullbackLeibler*.
lambda_max(exp)	largest eigenvalue of a square matrix expression (cf.
lambda_min(exp)	smallest eigenvalue of a square matrix expression (cf.
latin1([rebuildDerivedGlyphs])	Let PICOS create future string representations using ISO 8859-1 characters.
log(x)	The logarithm of a PICOS, CVXOPT, NumPy, or numeric expression.
logsumexp(exp)	A shorthand for *LogSumExp*.
lse(exp)	A shorthand for *LogSumExp*.
new_param(name, value)	Declare a parameter for the problem, that will be stored as a `cvxopt sparse matrix`.
norm(exp[, num, denom])	returns a *NormP_Exp* object representing the (generalized-) p-norm of the entries of `exp[:]`.
partial_trace(X[, k, dim])	Partial trace of an Affine Expression, with respect to the `k` th subsystem for a tensor product of dimensions `dim`.
partial_transpose(exp[, dims_1, subsystems, ...])	Partial transpose of an Affine Expression, with respect to given subsystems.
simplex([gamma])	returns a *TruncatedSimplex* object representing the set $\{x \geq 0 : \|\|x\|\|_1 \leq \gamma\}$.
sum(lst[, it, indices])	This is a replacement for Python's *sum* that produces sensible string representations when summing PICOS expressions.
sum_k_largest(exp, k)	returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest elements of an affine expression `exp`.
sum_k_largest_lambda(exp, k)	returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest eigenvalues of a square matrix affine expression `exp`.
sum_k_smallest(exp, k)	returns a *Sum_k_Smallest_Exp* object representing the sum of the `k` smallest elements of an affine expression `exp`.
sum_k_smallest_lambda(exp, k)	returns a *Sum_k_Smallest_Exp* object representing the sum of the `k` smallest eigenvalues of a square matrix affine expression `exp`.
sumexp(x[, y])	A shorthand for *SumExponential*.
trace(exp)	trace of a square AffinExp
tracepow(exp[, num, denom, coef])	Returns a *TracePow_Exp* object representing the trace of the pth-power of the symmetric matrix `exp`, where `exp` is an *AffinExp* which we denote by X.
truncated_simplex([gamma, sym])	returns a *TruncatedSimplex* object representing the set:
unicode([rebuildDerivedGlyphs])	Let PICOS create future string representations using unicode characters.

ascii

`picos.`**`ascii`**`()`

Let PICOS create future string representations using only ASCII characters.

ball

picos.**ball**(*r*, *p=2*)

returns a *Ball* object representing:

- a L_p Ball of radius r ($\{x : \|x\|_p \geq r\}$) if $p \geq 1$
- the convex set $\{x \geq 0 : \|x\|_p \geq r\}$ $p < 1$.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.ball(2,3)  #doctest: +NORMALIZE_WHITESPACE
<p-Norm Constraint: ‖x‖_3 ≤ 2>
>>> x << pic.ball(1,0.5)
<Generalized p-Norm Constraint: ‖x‖_(1/2) ≥ 1>
```

default_charset

picos.**default_charset**(*rebuildDerivedGlyphs=True*)

Let PICOS create future string representations using unicode characters.

detrootn

picos.**detrootn**(*exp*)

returns a *DetRootN_Exp* object representing the determinant of the n th-root of the symmetric matrix exp, where n is the dimension of the matrix. This can be used to enter constraints of the form $(\det X)^{1/n} \geq t$. Note that X is forced to be positive semidefinite when a constraint of this form is entered in PICOS. Determinant inequalities are internally reformulated as a set of Linear Matrix Inequalities (SDP).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> t < pic.detrootn(X)
<n-th Root of a Determinant Constraint: det(X)^(1/3) ≥ t>
```

diag

picos.**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.diag(x-y,4)
<4×4 Affine Expression: Diag(x - y)>
```

(continues on next page)

(continued from previous page)

```
>>> pic.diag(x//y)
<2×2 Affine Expression: Diag([x; y])>
```

diag_vect

picos.**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.diag_vect(X)
<3×1 Affine Expression: diag(X)>
```

exp

picos.**exp**(*x*)

Exponentiation of a PICOS, CVXOPT, NumPy, or numeric expression.

expcone

picos.**expcone**()

returns a *ExponentialCone* object representing the set closure $\{(x, y, z) : y > 0, y \exp(x/y) \leq z\}$

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> pic.expcone()
<Exponential Cone: cl{[x; y; z] : y·exp(z/y) ≤ x, x > 0, y > 0}>
>>> x << pic.expcone()
<Exponential Cone Constraint: x[0] ≥ x[1]·exp(x[2]/x[1])>
```

flow_Constraint

picos.**flow_Constraint**(*G*, *f*, *source*, *sink*, *flow_value*, *capacity=None*, *graphName=''*)

Constructs a network flow constraint.

Parameters

- **G** (networkx DiGraph.) – A directed graph.
- **f** (*dict*) – A dictionary of variables indexed by the edges of `G`.
- **source** – Either a node of `G` or a list of nodes in case of a multi-source flow.
- **sink** – Either a node of `G` or a list of nodes in case of a multi-sink flow.
- **flow_value** – The value of the flow, or a list of values in case of a single-source/multi-sink flow. In the latter case, the values represent the demands of each sink (resp. of each source for a multi-source/single-sink flow). The values can be either constants or *affine expressions*.
- **capacity** – Either `None` or a string. If this is a string, it indicates the key of the edge dictionaries of `G` that is used for the capacity of the links. Otherwise, edges have an unbounded capacity.
- **graphName** (*str*) – Name of the graph as used in the string representation of the constraint.

geomean

picos.**geomean**(*exp*)

returns a *GeoMeanExp* object representing the geometric mean of the entries of `exp[:]`. This can be used to enter inequalities of the form `t <= geomean(x)`. Note that geometric mean inequalities are internally reformulated as a set of SOC inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',1)
>>> y = prob.add_variable('y',3)
>>> # Add the constraint x <= (y0*y1*y2)**(1./3) to the problem:
>>> prob.add_constraint(x<pic.geomean(y))
<Geometric Mean Constraint: x ≤ geomean(y)>
```

get_version_info

picos.**get_version_info**()

import_cbf

picos.**import_cbf**(*filename*)

Imports the data from a CBF file, and creates a *Problem* object.

The created problem contains one (multidimmensional) variable for each cone specified in the section `VAR` of the .cbf file, and one (multidimmensional) constraint for each cone specified in the sections `CON` and `PSDCON`.

Semidefinite variables defined in the section `PSDVAR` of the .cbf file are represented by a matrix picos variable `X` with `X.vtype = 'symmetric'`.

This function returns a tuple `(P,x,X,data)`, where:

- `P` is the imported picos *Problem* object.
- `x` is a list of *Variable* objects, representing the (multidimmensional) scalar variables.
- `X` is a list of *Variable* objects, representing the symmetric semidefinite positive variables.
- `data` is a dictionary containing picos parameters (*AffinExp* objects) used to define the problem. Indexing is with respect to the blocks of variables as defined in the sections `VAR` and `CON` of the .cbf file.

kron

picos.**kron**(*A*, *B*)

Kronecker product of 2 expression, at least one of which must be constant

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> import numpy as np
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,3))
>>> X.value = cvx.matrix(range(12),(4,3))
>>> I = pic.new_param('I',np.eye(2))
>>> print(pic.kron(I,X)) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  0.00e+00  0.00e+00  0.00e+00]
[ 1.00e+00  5.00e+00  9.00e+00  0.00e+00  0.00e+00  0.00e+00]
[ 2.00e+00  6.00e+00  1.00e+01  0.00e+00  0.00e+00  0.00e+00]
```

(continues on next page)

(continued from previous page)

```
[ 3.00e+00  7.00e+00  1.10e+01  0.00e+00  0.00e+00  0.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  4.00e+00  8.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  1.00e+00  5.00e+00  9.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  2.00e+00  6.00e+00  1.00e+01]
[ 0.00e+00  0.00e+00  0.00e+00  3.00e+00  7.00e+00  1.10e+01]
```

kullback_leibler

picos.**kullback_leibler**(*x*, *y=None*)

A shorthand for *KullbackLeibler*.

If the second optional argument is passed, the resulting expression is the Kullback-Leibler divergence $\sum_i x_i \log(x_i/y_i)$, otherwise it is the (negative) entropy $\sum_i x_i \log(x_i)$.

lambda_max

picos.**lambda_max**(*exp*)

largest eigenvalue of a square matrix expression (cf. *pic.sum_k_largest(exp,1)*)

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> pic.lambda_max(X) < 2
<Sum of Largest Eigenvalues Constraint: λ_max(X) ≤ 2>
```

lambda_min

picos.**lambda_min**(*exp*)

smallest eigenvalue of a square matrix expression (cf. *pic.sum_k_smallest(exp,1)*)

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> pic.lambda_min(X) > -1
<Sum of Smallest Eigenvalues Constraint: λ_min(X) ≥ -1>
```

latin1

picos.**latin1**(*rebuildDerivedGlyphs=True*)

Let PICOS create future string representations using ISO 8859-1 characters.

log

picos.**log**(*x*)

The logarithm of a PICOS, CVXOPT, NumPy, or numeric expression.

logsumexp

picos.**logsumexp**(*exp*)

A shorthand for *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]]))
```

(continues on next page)

(continued from previous page)

```
>>> pic.lse(A*x)<0
<LSE Constraint: logsumoexp(A·x) ≤ 0>
```

lse

picos.**lse**(*exp*)

A shorthand for *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
<LSE Constraint: logsumoexp(A·x) ≤ 0>
```

new_param

picos.**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 picos as pic
>>> 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']
<2×1 Affine Expression: [2×3]·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']
<2×1 Affine Expression: B[matrix]·x + [B[foo]]>
```

norm

picos.**norm**(*exp*, *num=2*, *denom=1*)

returns a *NormP_Exp* object representing the (generalized-) p-norm of the entries of `exp[:]`. This can be used to enter constraints of the form $\|x\|_p \leq t$ with $p \geq 1$. Generalized norms are also defined for $p < 1$, by using the usual formula $\text{norm}(x,p) := \left(\sum_i x_i^p\right)^{1/p}$. Note that this function is concave (for $p < 1$) over the set of vectors with nonnegative coordinates. When a constraint of the form $\text{norm}(x,p) > t$ with $p \leq 1$ is entered, PICOS implicitely assumes that x is a nonnegative vector.

This function can also be used to represent the Lp,q- norm of a matrix (for $p, q \geq 1$): $\text{norm}(X,(p,q)) := \left(\sum_i (\sum_j x_{ij}^q)^{p/q}\right)^{1/p}$, that is, the p-norm of the vector formed with the q-norms of the rows of X.

The exponent p of the norm must be specified either by a couple numerator (2d argument) / denominator (3d arguments), or directly by a float `p` given as second argument. In the latter case a rational approximation of `p` will be used. It is also possible to pass `'inf'` as second argument for the infinity-norm (aka max-norm).

For the case of (p,q)-norms, `p` and `q` must be specified by a tuple of floats in the second argument (rational approximations will be used), and the third argument will be ignored.

Example:

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x',1)
>>> y = P.add_variable('y',3)
>>> pic.norm(y,7,3) < x
<p-Norm Constraint: ‖y‖_(7/3) ≤ x>
>>> pic.norm(y,-0.4) > x
<Generalized p-Norm Constraint: ‖y‖_(-2/5) ≥ x>
>>> X = P.add_variable('X',(3,2))
>>> pic.norm(X,(1,2)) < 1
<(p,q)-Norm Constraint: ‖X‖_1,2 ≤ 1>
>>> pic.norm(X,('inf',1)) < 1
<(p,q)-Norm Constraint: ‖X‖_inf,1 ≤ 1>
```

partial_trace

picos.**partial_trace**(*X*, *k=1*, *dim=None*)

Partial trace of an Affine Expression, with respect to the `k` th subsystem for a tensor product of dimensions `dim`. If `X` is a matrix *AffinExp* that can be written as $X = A_0 \otimes \cdots \otimes A_{n-1}$ for some matrices $A_0, \ldots, A_{n-1}$ of respective sizes `dim[0] x dim[0]`, ... , `dim[n-1] x dim[n-1]` (`dim` is a list of ints if all matrices are square), or `dim[0][0] x dim[0][1]`, ...,‘‘dim[n-1][0] x dim[n-1][1]‘‘ (`dim` is a list of 2-tuples if any of them except the `k` th one is rectangular), this function returns the matrix $Y = \text{trace}(A_k) \quad A_0 \otimes \cdots A_{k-1} \otimes A_{k+1} \otimes \cdots \otimes A_{n-1}$.

The default value `dim=None` automatically computes the size of the subblocks, assuming that `X` is a $n^2 \times n^2$-square matrix with blocks of size $n \times n$.

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,4))
>>> X.value = cvx.matrix(range(16),(4,4))
>>> print(X) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
[ 1.00e+00  5.00e+00  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  1.00e+01  1.40e+01]
[ 3.00e+00  7.00e+00  1.10e+01  1.50e+01]
>>> # Partial trace with respect to second subsystem (k=1):
```

(continues on next page)

(continued from previous page)

```
>>> print(pic.partial_trace(X)) #doctest: +NORMALIZE_WHITESPACE
[ 5.00e+00  2.10e+01]
[ 9.00e+00  2.50e+01]
>>> # And with respect to first subsystem (k=0):
>>> print(pic.partial_trace(X,0)) #doctest: +NORMALIZE_WHITESPACE
[ 1.00e+01  1.80e+01]
[ 1.20e+01  2.00e+01]
```

partial_transpose

picos.**partial_transpose**(*exp*, *dims_1=None*, *subsystems=None*, *dims_2=None*)

Partial transpose of an Affine Expression, with respect to given subsystems. If `X` is a matrix *AffinExp* that can be written as $X = A_0 \otimes \cdots \otimes A_{n-1}$ for some matrices $A_0, \ldots, A_{n-1}$ of respective sizes `dims_1[0] x dims_2[0], ... , dims_1[n-1] x dims_2[n-1]`, this function returns the matrix $Y = B_0 \otimes \cdots \otimes B_{n-1}$, where $B_i = A_i^T$ if `i in subsystems`, and $B_i = A_i$ otherwise.

The optional parameters `dims_1` and `dims_2` are tuples specifying the dimension of each subsystem A_i. The argument `subsystems` must be a `tuple` (or an `int`) with the index of all subsystems to be transposed.

The default value `dims_1=None` automatically computes the size of the subblocks, assuming that `exp` is a $n^k \times n^k$-square matrix, for the *smallest* appropriate value of $k \in [2, 6]$, that is `dims_1=(n,)*k`.

If `dims_2` is not specified, it is assumed that the subsystems A_i are square, i.e., `dims_2=dims_1`. If `subsystems` is not specified, the default assumes that only the last system must be transposed, i.e., `subsystems = (len(dims_1)-1,)`

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,4))
>>> X.value = cvx.matrix(range(16),(4,4))
>>> print(X) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
[ 1.00e+00  5.00e+00  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  1.00e+01  1.40e+01]
[ 3.00e+00  7.00e+00  1.10e+01  1.50e+01]
>>> # Standard partial transpose with respect to the 2x2 blocks and 2nd␣
↪subsystem:
>>> print(X.Tx) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  1.00e+00  8.00e+00  9.00e+00]
[ 4.00e+00  5.00e+00  1.20e+01  1.30e+01]
[ 2.00e+00  3.00e+00  1.00e+01  1.10e+01]
[ 6.00e+00  7.00e+00  1.40e+01  1.50e+01]
>>> # Now with respect to the first subsystem:
>>> print(pic.partial_transpose(X,(2,2),0)) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  2.00e+00  6.00e+00]
[ 1.00e+00  5.00e+00  3.00e+00  7.00e+00]
[ 8.00e+00  1.20e+01  1.00e+01  1.40e+01]
[ 9.00e+00  1.30e+01  1.10e+01  1.50e+01]
```

simplex

picos.**simplex**(*gamma=1*)

returns a *TruncatedSimplex* object representing the set $\{x \geq 0 : ||x||_1 \leq \gamma\}$.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.simplex()
<Standard Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 1}>
>>> x << pic.simplex(2)
<Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 2}>
```

sum

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

This is a replacement for Python's *sum* that produces sensible string representations when summing PICOS expressions.

Parameters

- **lst** – A list of *expressions*.
- **it** – DEPRECATED
- **indices** – DEPRECATED

Example:

```
>>> import picos
>>> P = picos.Problem()
>>> x = P.add_variable("x", 5)
>>> e = [x[i]*x[i+1] for i in range(len(x) - 1)]
>>> sum(e)
<Quadratic Expression: x[0]·x[1] + x[1]·x[2] + x[2]·x[3] + x[3]·x[4]>
>>> picos.sum(e)
<Quadratic Expression: ∑(x[i]·x[i+1] : i ∈ [0...3])>
```

sum_k_largest

picos.**sum_k_largest**(*exp*, *k*)

returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest elements of an affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} x_i^{\downarrow} \leq t$. This kind of constraints is reformulated internally as a set of linear inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',3)
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_largest(x,2) < 1
<Sum of Largest Elements Constraint: sum_2_largest(x) ≤ 1>
>>> pic.sum_k_largest(x,1) < t
<Sum of Largest Elements Constraint: max(x) ≤ t>
```

sum_k_largest_lambda

picos.**sum_k_largest_lambda**(*exp*, *k*)

returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest eigenvalues of a square matrix affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} \lambda_i^{\downarrow}(X) \leq t$. This kind of constraints is reformulated internally as a set of linear matrix inequalities (SDP). Note that `exp` is assumed to be symmetric (picos does not check).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_largest_lambda(X,3) < 1
<Sum of Largest Eigenvalues Constraint: trace(X) ≤ 1>
>>> pic.sum_k_largest_lambda(X,2) < t
<Sum of Largest Eigenvalues Constraint: sum_2_largest_λ(X) ≤ t>
```

sum_k_smallest

picos.**sum_k_smallest**(*exp*, *k*)

returns a *Sum_k_Smallest_Exp* object representing the sum of the k smallest elements of an affine expression exp. This can be used to enter constraints of the form $\sum_{i=1}^{k} x_i^{\uparrow} \geq t$. This kind of constraints is reformulated internally as a set of linear inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',3)
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_smallest(x,2) > t
<Sum of Smallest Elements Constraint: sum_2_smallest(x) ≥ t>
>>> pic.sum_k_smallest(x,1) > 3
<Sum of Smallest Elements Constraint: min(x) ≥ 3>
```

sum_k_smallest_lambda

picos.**sum_k_smallest_lambda**(*exp*, *k*)

returns a *Sum_k_Smallest_Exp* object representing the sum of the k smallest eigenvalues of a square matrix affine expression exp. This can be used to enter constraints of the form $\sum_{i=1}^{k} \lambda_i^{\uparrow}(X) \geq t$. This kind of constraints is reformulated internally as a set of linear matrix inequalities (SDP). Note that exp is assumed to be symmetric (picos does not check).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_smallest_lambda(X,1) > 1
<Sum of Smallest Eigenvalues Constraint: λ_min(X) ≥ 1>
>>> pic.sum_k_smallest_lambda(X,2) > t
<Sum of Smallest Eigenvalues Constraint: sum_2_smallest_λ(X) ≥ t>
```

sumexp

picos.**sumexp**(*x*, *y=None*)

A shorthand for *SumExponential*.

If the second optional argument is passed, the resulting expression is the sum of perspectives of exponentials $\sum_i y_i \exp(x_i/y_i)$, otherwise it is a sum of exponentials $\sum_i \exp(x_i)$.

trace

picos.**trace**(*exp*)

trace of a square AffinExp

tracepow

picos.**tracepow**(*exp*, *num=1*, *denom=1*, *coef=None*)

Returns a *TracePow_Exp* object representing the trace of the pth-power of the symmetric matrix `exp`, where `exp` is an *AffinExp* which we denote by X. This can be used to enter constraints of the form $\text{trace}\, X^p \leq t$ with $p \geq 1$ or $p < 0$, or $\text{trace}\, X^p \geq t$ with $0 \leq p \leq 1$. Note that X is forced to be positive semidefinite when a constraint of this form is entered in PICOS.

It is also possible to specify a `coef` matrix (M) of the same size as `exp`, in order to represent the expression $\text{trace}(MX^p)$. The constraint $\text{trace}(MX^p) \geq t$ can be reformulated with SDP constraints if M is positive semidefinite and $0 < p < 1$.

Trace of power inequalities are internally reformulated as a set of Linear Matrix Inequalities (SDP), or second order cone inequalities if `exp` is a scalar.

The exponent p of the norm must be specified either by a couple numerator (2d argument) / denominator (3d arguments), or directly by a float `p` given as second argument. In the latter case a rational approximation of `p` will be used.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.tracepow(X,7,3) < t
<Trace of Power Constraint: trace(X^(7/3)) ≤ t>
>>> pic.tracepow(X,0.6) > t
<Trace of Power Constraint: trace(X^(3/5)) ≥ t>
>>> import cvxopt as cvx
>>> A = cvx.normal(3,3);A=A*A.T # A random semidefinite positive matrix
>>> A = pic.new_param('A',A)
>>> pic.tracepow(X,0.25,coef=A) > t
<Trace of Power Constraint: trace(A·X^(1/4)) ≥ t>
```

truncated_simplex

picos.**truncated_simplex**(*gamma=1*, *sym=False*)

returns a *TruncatedSimplex* object representing the set:

- $\{x \geq 0 : ||x||_\infty \leq 1,\ ||x||_1 \leq \gamma\}$ if `sym=False` (default)
- $\{x : ||x||_\infty \leq 1,\ ||x||_1 \leq \gamma\}$ if `sym=True`.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.truncated_simplex(2)
<Truncated Simplex Constraint: x ∈ {0 ≤ x ≤ 1 : ∑(x) ≤ 2}>
>>> x << pic.truncated_simplex(2,sym=True)
<Symmetrized Truncated Simplex Constraint: x ∈ {-1 ≤ x ≤ 1 : ∑(|x|) ≤ 2}>
```

unicode

picos.**unicode**(*rebuildDerivedGlyphs=True*)

Let PICOS create future string representations using unicode characters.

Classes

DualizationError(msg)	Exception raised when a non-standard conic problem is being dualized.
NonConvexError(msg)	Exception raised when non-convex quadratic constraints are passed to a solver which cannot handle them.
NotAppropriateSolverError(msg)	Exception raised when trying to solve a problem with a solver which cannot handle it
Problem(**options)	PICOS' representation of an optimization problem.
QuadAsSocpError(msg)	Exception raised when the problem can not be solved in the current form, because quad constraints are not handled.
new_problem	alias of *picos.problem.Problem*

DualizationError

exception picos.**DualizationError**(*msg*)
Bases: Exception

Exception raised when a non-standard conic problem is being dualized.

NonConvexError

exception picos.**NonConvexError**(*msg*)
Bases: Exception

Exception raised when non-convex quadratic constraints are passed to a solver which cannot handle them.

NotAppropriateSolverError

exception picos.**NotAppropriateSolverError**(*msg*)
Bases: Exception

Exception raised when trying to solve a problem with a solver which cannot handle it

Problem

class picos.**Problem**(**options*)
Bases: object

PICOS' representation of an optimization problem.

Example

```
>>> import picos
>>> P = picos.Problem(verbose = 0)
>>> X = P.add_variable("X", (2,2), lower = 0)
>>> # 1|X is the dot prouct of X with a matrix of all ones.
>>> C = P.add_constraint(1|X < 10)
>>> P.set_objective("max", picos.trace(X))
>>> # PICOS will select a suitable solver if you don't specify one.
>>> solution = P.solve(solver = "cvxopt")
>>> solution["status"]
'optimal'
>>> solution["time"] #doctest: +SKIP
0.00034999847412109375
>>> round(P.obj_value(), 1)
10.0
>>> print(X) #doctest: +SKIP
[ 0.00e+00  0.00e+00]
[ 0.00e+00  1.00e+01]
```

(continues on next page)

(continued from previous page)

```
>>> round(C.dual, 1)
1.0
```

Creates an empty problem and optionally sets initial solver options.

Parameters **options** – A parameter sequence of solver options.

Attributes Summary

options	
status	The solution status of the problem.
type	The problem type as a string, such as 'LP', 'MILP' or 'SOCP'.

Methods Summary

add_constraint(constraint[, key])	Adds a constraint to the problem.
add_list_of_constraints(lst[, it, indices, key])	Adds a list of constraints to the problem, enabling the use of Python list comprehensions (see the example below).
add_variable(name[, size, vtype, lower, upper])	Adds a variable to the problem and returns it for use in constraints.
as_dual()	Returns the Lagrangian dual problem of the problem.
as_real()	Returns a modified copy of the problem, where hermitian nn matrices are replaced by symmetric $2n2n$ matrices.
as_socp()	
check_current_value_feasibility([tol, inttol])	Checks whether all variables that appear in constraints are valued and satisfy the constraints up to the given tolerance.
convert_quad_to_socp()	Replaces quadratic constraints with equivalent second order cone constraints.
convert_quadobj_to_constraint()	Replaces a quadratic objective with an equivalent quadratic constraint.
copy()	Creates an independent copy of the problem, using new variables.
get_constraint(ind)	Returns a constraint of the problem, given its index.
get_valued_variable(name)	Returns the value or values of a variable or of a collection of variables with a common base name.
get_variable(name)	Returns a single variable with the given name or a list or dictionary of variables with the given name as a common base name.
is_complex()	**returns** `True`, if the problem has a complex variable or if there
is_continuous()	**returns** `True`, if all variables are continuous.

Continued on next page

Table 4 – continued from previous page

is_pure_integer()	**returns** `True`, if all variables are integer.
maximize(obj, **options)	Sets the objective to maximize the given objective function and calls the solver with the given sequence of options.
minimize(obj, **options)	Sets the objective to minimize the given objective function and calls the solver with the given sequence of options.
obj_value()	Returns the objective value after the problem was solved.
remove_all_constraints()	Removes all constraints from the problem.
remove_all_variable_bounds()	Removes all lower and upper bounds from all variables.
remove_constraint(ind)	Deletes a constraint from the problem.
remove_variable(name)	Removes a variable from the problem.
reset([resetOptions])	Resets the problem instance to its initial empty state.
reset_solver_instances()	Resets all solver instances, so that the problem will be reimported and solved from scratch.
set_all_options_to_default()	Sets all solver options to their default value.
set_objective(typ, expr)	Sets the objective function and optimization direction of the problem.
set_option(key, val)	Sets a single solver option to the given value.
set_var_value(name, value[, optimalvar])	Sets the *value* of the given variable.
solve(**options)	Hands the problem to a solver.
update_options(**options)	Sets multiple solver options at once.
verbosity()	**returns** The problem's current verbosity level.
write_to_file(filename[, writer])	Writes the problem to a file.

Attributes Documentation

options

status
The solution status of the problem.

type
The problem type as a string, such as 'LP', 'MILP' or 'SOCP'.

Methods Documentation

add_constraint(*constraint*, *key=None*)
Adds a constraint to the problem.

Parameters

- **constraint** (*Constraint*) – The constraint to be added.
- **key** (*str*) – Optional name of the constraint.

Returns The constraint that was added to the problem, which may be a *MetaConstraint* that contains further references to auxiliary constraints that were also added, as well as potentially references to new auxiliary variables.

add_list_of_constraints(*lst*, *it=None*, *indices=None*, *key=None*)
Adds a list of constraints to the problem, enabling the use of Python list comprehensions (see the

example below).

Parameters

- **lst** (*list*) – A list of *constraints*.
- **it** – DEPRECATED
- **indices** – DEPRECATED
- **key** (*str*) – A name describing the list of constraints.

Returns A list of all constraints that were added.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> pprint(x)
[<2×1 Continuous Variable: x[0]>,
 <2×1 Continuous Variable: x[1]>,
 <2×1 Continuous Variable: x[2]>,
 <2×1 Continuous Variable: x[3]>,
 <2×1 Continuous Variable: x[4]>]
>>> y=prob.add_variable('y',5)
>>> IJ=[(1,2),(2,0),(4,2)]
>>> w={}
>>> for ij in IJ:
...         w[ij]=prob.add_variable('w[{},{}]'.format(*ij),3)
...
>>> u=pic.new_param('u',cvx.matrix([2,5]))
>>> C1=prob.add_list_of_constraints([u.T*x[i] < y[i] for i in range(5)])
>>> C2=prob.add_list_of_constraints([abs(w[i,j])<y[j] for (i,j) in IJ])
>>> C3=prob.add_list_of_constraints([y[t] > y[t+1] for t in range(4)])
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (SOCP):
24 variables, 9 affine constraints, 12 vars in 3 SO cones
<BLANKLINE>
w   : dict of 3 variables, (3, 1), continuous
x   : list of 5 variables, (2, 1), continuous
y   : (5, 1), continuous
<BLANKLINE>
    find vars
such that
  u.T·x[i] ≤ y[i] ∀ i ∈ [0...4]
  ‖w[i,j]‖ ≤ y[j] ∀ (i,j) ∈ zip([1,2,4],[2,0,2])
  y[i] ≥ y[i+1] ∀ i ∈ [0...3]
---------------------
```

add_variable(*name*, *size=1*, *vtype='continuous'*, *lower=None*, *upper=None*)

Adds a variable to the problem and returns it for use in constraints.

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 an n-dimensional vector,
  - or a `tuple` (n, m), in which case the variable is a nm matrix.
- **vtype** (*str*) – Domain of the variable. Can be any of

- `'continuous'` – real valued,
- `'binary'` – either zero or one,
- `'integer'` – integer valued,
- `'symmetric'` – symmetric matrix,
- `'antisym'` – antisymmetric matrix,
- `'complex'` – complex matrix,
- `'hermitian'` – complex hermitian matrix,
- `'semicont'` – zero or real valued and satisfying its bounds (supported by CPLEX and Gurobi only), or
- `'semiint'` – zero or integer valued and satisfying its bounds (supported by CPLEX and Gurobi only).

- **lower** (anything recognized by *retrieve_matrix*) – A lower bound for the variable.

  Can be either a vector or matrix of the same size as the variable or a scalar that is then broadcasted so that all elements of the variable have the same lower bound.

- **upper** (anything recognized by *retrieve_matrix*) – An upper bound for the variable.

  Can be either a vector or matrix of the same size as the variable or a scalar that is then broadcasted so that all elements of the variable have the same upper bound.

Returns A *Variable* instance.

Example

```
>>> prob=picos.Problem()
>>> x=prob.add_variable('x',3)
>>> x
<3×1 Continuous Variable: x>
```

as_dual()
Returns the Lagrangian dual problem of the problem.

To this end the problem is put in a canonical primal form (see the *note on dual variables*), and the corresponding dual form is returned as a new *Problem*.

as_real()
Returns a modified copy of the problem, where hermitian nn matrices are replaced by symmetric $2n2n$ matrices.

as_socp()

check_current_value_feasibility(*tol=1e-05*, *inttol=0.001*)
Checks whether all variables that appear in constraints are valued and satisfy the constraints up to the given tolerance. In other words, checks whether the variables are valued to form a feasible solution.

Parameters

- **tol** (*float*) – Largest tolerated absolute violation of a constraint. If `None`, the `fesatol` or `tol` solver option is used.
- **inttol** (*float*) – Largest tolerated absolute violation of integrality of an integral variable.

Returns A tuple `(feasible, violation)` where `feasible` is a bool stating whether the solution is feasible and `violation` is either `None`, if `feasible == True`, or the amount of violation, otherwise.

convert_quad_to_socp()
Replaces quadratic constraints with equivalent second order cone constraints.

convert_quadobj_to_constraint()
Replaces a quadratic objective with an equivalent quadratic constraint.

copy()
Creates an independent copy of the problem, using new variables.

Note: Your existing variable, constraint, and metaconstraint references will refer to the original variables, so you cannot query these for solution details after solving the copy. Access the copy's constraints and variables instead.

---

get_constraint(*ind*)
Returns a constraint of the problem, given its index.

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

- If `ind` is an `int` n, then the n-th constraint (starting from 0) is returned, where constraints are counted in the order in which they where passed to the problem.
- if `ind` is a `tuple` (k, i), then the i-th constraint from the k-th group of constraints is returned (both starting from 0). Here *group of constraints* refers to a list of constraints added together via *add_list_of_constraints*.
- If `ind` is a tuple $(k,)$ of length 1, then the k-th group of constraints is returned as a list.

Returns A *constraint* or a list thereof.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> y=prob.add_variable('y',5)
>>> Cx=prob.add_list_of_constraints([(1|x[i]) < y[i] for i in range(5)])
>>> Cy=prob.add_constraint(y>0)
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (LP):
15 variables, 10 affine constraints
<BLANKLINE>
x   : list of 5 variables, (2, 1), continuous
y   : (5, 1), continuous
<BLANKLINE>
    find vars
such that
  ⟨[1], x[i]⟩ ≤ y[i] ∀ i ∈ [0...4]
  y ≥ 0
---------------------
>>> # Retrieve the 3nd constraint (counted from 0):
>>> prob.get_constraint(1)
<1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>
>>> # Retrieve the 4th consraint from the 1st group:
>>> prob.get_constraint((0,3))
<1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>
>>> # Retrieve the unique constraint of the 2nd 'group':
>>> prob.get_constraint((1,))
<5×1 Affine Constraint: y ≥ 0>
>>> # Retrieve the whole 1st group of constraints:
```

(continues on next page)

(continued from previous page)

```
>>> pprint(prob.get_constraint((0,)))
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <1×1 Affine Constraint: ⟨[1], x[4]⟩ ≤ y[4]>]
```

get_valued_variable(*name*)

Returns the value or values of a variable or of a collection of variables with a common base name.

Parameters **name** (*str*) – Name of a single variable or of a collection of variables (see *get_variable* on how to specify collections).

Raises An exception if any of the variables is not valued, in particular when the problem was not yet solved.

Returns A CVXOPT matrix, if name refers to a single variable, or a list or a dictionary thereof, if the collection of variables specified by name is a list or a dictionary, respectively.

get_variable(*name*)

Returns a single variable with the given name or a list or dictionary of variables with the given name as a common base name. In the latter case the variables must be named name[index] or name[key] with index taken from a set of integer strings and key taken from a set of arbitrary strings.

Parameters **name** (*str*) – Name of a single variable or of a collection of variables.

Returns A *PICOS variable*, if name refers to a single variable, or a list or a dictionary thereof, if the collection of variables specified by name is a list or a dictionary, respectively.

is_complex()

Returns True, if the problem has a complex variable or if there is a complex coefficient or constant inside a constraint.

is_continuous()

Returns True, if all variables are continuous.

is_pure_integer()

Returns True, if all variables are integer.

maximize(*obj*, ***options*)

Sets the objective to maximize the given objective function and calls the solver with the given sequence of options.

Parameters

- **obj** (*Expression*) – The objective function to maximize.
- **options** – A sequence of optional solver options.

Returns A dictionary, see *solve*.

> **Warning:** This is equivalent to *set_objective* followed by *solve* and will thus override any existing objective function and direction.
>
> Further, any supplied options will be stored in the problem as if they were set via *set_option*.

minimize(*obj*, ***options*)

Sets the objective to minimize the given objective function and calls the solver with the given sequence of options.

Parameters

- **obj** (*Expression*) – The objective function to minimize.
- **options** – A sequence of optional solver options.

Returns A dictionary, see *solve*.

> **Warning:** This is equivalent to *set_objective* followed by *solve* and will thus override any existing objective function and direction.
>
> Further, any supplied options will be stored in the problem as if they were set via *set_option*.

obj_value()
Returns the objective value after the problem was solved.

Raises AttributeError, if the problem was not yet solved.

remove_all_constraints()
Removes all constraints from the problem.

This function does not remove bounds set directly on variables; use *remove_all_variable_bounds* to do so.

remove_all_variable_bounds()
Removes all lower and upper bounds from all variables.

remove_constraint(*ind*)
Deletes a constraint from the problem.

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

- If ind is an int n, then the n-th constraint (starting from 0) is deleted, where constraints are counted in the order in which they where passed to the problem.
- if ind is a tuple (k, i), then the i-th constraint from the k-th group of constraints is deleted (both starting from 0). Here *group of constraints* refers to a list of constraints added together via *add_list_of_constraints*.
- If ind is a tuple $(k,)$ of length 1, then the whole k-th group of constraints is deleted.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(4)]
>>> y=prob.add_variable('y',4)
>>> Cxy=prob.add_list_of_constraints([(1|x[i])<y[i] for i in range(4)])
>>> Cy=prob.add_constraint(y>0)
>>> Cx0to2=prob.add_list_of_constraints([x[i]<2 for i in range(3)])
>>> Cx3=prob.add_constraint(x[3]<1)
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <4×1 Affine Constraint: y ≥ 0>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 2nd constraint (counted from 0):
>>> prob.remove_constraint(1)
```

(continues on next page)

(continued from previous page)

```
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <4×1 Affine Constraint: y ≥ 0>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 2nd group of constraints, i.e. the constraint y > 0:
>>> prob.remove_constraint((1,))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 3rd remaining group of constraints, i.e. x[3] < [1]:
>>> prob.remove_constraint((2,))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>]
>>> # Delete 2nd constraint of the 2nd remaining group, i.e. x[1] < |2|:
>>> prob.remove_constraint((1,1))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>]
```

remove_variable(*name*)
Removes a variable from the problem.

Parameters **name** (*str*) – Name of the variable to remove.

Warning: This method does not check if some constraint still involves the variable to be removed.

reset(*resetOptions=False*)
Resets the problem instance to its initial empty state.

Parameters **resetOptions** (*bool*) – Whether also solver options should be reset to their default values.

reset_solver_instances()
Resets all solver instances, so that the problem will be reimported and solved from scratch.

set_all_options_to_default()
Sets all solver options to their default value.

set_objective(*typ*, *expr*)
Sets the objective function and optimization direction of the problem.

Parameters

- **typ** (*str*) – Can be either `'max'`, `'min'`, or `'find'`, for a maximization, minimization, and feasibility problem, respectively.
- **expr** (*Expression*) – The objective function to be minimized or maximized. This parameter is ignored if `typ == 'find'`.

set_option(*key*, *val*)

Sets a single solver option to the given value.

Parameters

- **key** (*str*) – String name of the option, see below for a list.
- **val** – New value for the option.

The following options are available and are listed with their default values.

- General options common to all solvers:
  - `strict_options = False` – If `True`, unsupported general options will raise an *UnsupportedOptionError* exception, instead of printing a warning.
  - `verbose = 1` – Verbosity level.
    - `-1` attempts to suppress all output, even errors.
    - `0` only outputs warnings and errors.
    - `1` generates standard informative output.
    - `2` prints all available information for debugging purposes.
  - `allow_license_warnings = True` – Whether solvers are allowed to ignore the `verbose` option to print licensing related warnings.

    Using this option to surpress licensing related warnings is done at your own legal responsibility.
  - `solver = None` – Solver to use.
    - `None` lets PICOS select a suitable solver for you.
    - `'cplex'` for CPLEX.
    - `'cvxopt'` for CVXOPT.
    - `'glpk'` for GLPK.
    - `'mosek'` for MOSEK.
    - `'gurobi'` for Gurobi.
    - `'scip'` for SCIP (formerly ZIBOpt).
    - `'smcp'` for SMCP.
  - `tol = 1e-8` – Relative gap termination tolerance for interior-point optimizers (feasibility and complementary slackness).

    This option is currently ignored by GLPK. SCIP will only lower its precision for large values and not increase it for small ones.
  - `maxit = None` – Maximum number of iterations for simplex or interior-point optimizers). *Currently ignored by SCIP.*
  - `lp_root_method = None` – Algorithm used to solve continuous linear problems, including the root relaxation of mixed integer problems.
    - `None` lets PICOS or the solver select it fo you.
    - `'psimplex'` for primal Simplex.
    - `'dsimplex'` for dual Simplex.

  * 'interior' for the interior point method.

  This option currently works only with CPLEX, Gurobi and MOSEK. With GLPK it works for LPs but not for the MIP root relaxation.

- lp_node_method = None – Algorithm used to solve subproblems at non-root nodes of the branching tree built when solving mixed integer programs.

  * None lets PICOS or the solver select it fo you.

  * 'psimplex' for primal Simplex.

  * 'dsimplex' for dual Simplex.

  * 'interior' for the interior point method.

  This option currently works only with CPLEX, Gurobi and MOSEK.

- timelimit = None – Total time limit for the solver, in seconds. The default None means no time limit.

  This option is not supported by CVXOPT and SMCP.

- treememory = None – Bound on the memory used by the branch and bound tree, in Megabytes.

  This option currently works only with CPLEX and SCIP.

- gaplim = 1e-4 – For mixed integer problems, the solver returns a solution as soon as this value for the relative gap between the primal and the dual bound is reached.

- noprimals = False – If True, do not retrieve a primal solution from the solver.

- noduals = False – If True, do not retrieve optimal values for the dual variables. This can speed up solvers that do not produce a dual solution as part of their primal solution process.

- nbsol = None – Maximum number of feasible solution nodes visited when solving a mixed integer problem, before returning the best one found.

  If you want to obtain all feasible solutions that the solver encountered, use pool_size instead.

- pool_size = None – Maximum number of mixed integer feasible solutions returned, instead of just a single one.

  If you merely want to set a limit on the number of feastble solution nodes that are visited, use nbsol instead.

  This option currently works only with CPLEX.

- pool_absgap = None – Discards solutions from the solution pool as soon as a better solution is found that beats it by the given absolute gap tolerance with respect to the objective function.

  This option currently works only with CPLEX.

- pool_relgap = None – Discards solutions from the solution pool as soon as a better solution is found that beats it by the given relative gap tolerance with respect to the objective function.

  This option currently works only with CPLEX.

- hotstart = False – If *True*, tells the mixed integer optimizer to start from the (partial) solution specified in the variables' *value* attributes.

  This option currently works only with CPLEX, Gurobi and MOSEK.

- solve_via_dual = None – If set to True, the Lagrangian Dual (computed with the function *as_dual*) is passed to the solver, instead of the problem itself. In some scenarios

this can yield a signficant speed-up. If set to `None`, PICOS chooses automatically whether the problem itself or its dual should be passed to the solver.

- 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`.
  - `uboundlimit = None` – Tells CPLEX to stop as soon as an upper bound smaller than this value is found.
  - `lboundlimit = None` – Tells CPLEX to stop as soon as a lower bound larger than this value is found.
  - `boundMonitor = True` – Tells CPLEX to store information about the evolution of the bounds during the solving process. At the end of the computation, a list of triples *(time,lowerbound,upperbound)* will be provided in the field bounds_monitor‘ of the dictionary returned by *solve*.
- Specific options available for CVXOPT, SMCP and ECOS:
  - `feastol = None` – Feasibility tolerance passed to cvx.solvers.options If `None`, then the value of the option `tol` is used.
  - `abstol = None` – Absolute tolerance passed to cvx.solvers.options If `None`, then the value of the option `tol` is used.
  - `reltol = None` – relative tolerance passed to cvx.solvers.options If `None`, then **ten times** the value of the option `tol` is used.
- 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.
- Specific options available for MOSEK:
  - `mosek_params = {}` – A dictionary of MOSEK Fusion API parameters to be set before the MOSEK optimizer is called.
- Specific options available for SCIP:
  - `scip_params = {}` – A dictionary of SCIP parameters to be set before the SCIP optimizer is called.

    For example, `scip_params = {'lp/threads': 4}` sets the number of threads to solve LPs with to `4`.

Note: Options can also be passed as a parameter sequence of the form `key = value` when the *Problem* is created or later to the function *solve*.

---

set_var_value(*name*, *value*, *optimalvar=False*)

Sets the *value* of the given variable.

Parameters

- **or tuple name** (*str*) – Name of the variable.
- **value** (Anything recognized by *retrieve_matrix*) – The value to be set.

Example

```
>>> prob=picos.Problem()
>>> x=prob.add_variable('x', 2)
>>> prob.set_var_value('x', [3,4]) # equivalent to x.value = [3,4]
>>> abs(x)**2
<Quadratic Expression: ‖x‖^2>
>>> print(abs(x)**2)
25.0
```

Note: The `hotstart` option allows certain solvers to leverage variables that were valued manually or by a preceding solution search.

solve(***options*)
Hands the problem to a solver.

You can select the solver manually with the `solver` option. Otherwise a suitable solver will be selected among those that are available on the platform.

Once the problem has been solved, the optimal solution can be obtained by querying the *value* property of the variables and the optimal dual values can be accessed via the *dual* property of the constraints.

Parameters **options** – A sequence of optional solver options. In particular, you can use this to select a solver via the `solver` option.

Returns

A dictionary that contains the following common entries, and potentially further solver-specific or option-specific fields:

- `'status'` – The solution status as a human readable string, such as `'optimal'` or `'infeasible'`. The exact wording and available phrases depend on the solver being used.
- `'time'` – The time spent searching for a solution in seconds, *excluding* any overhead produced by PICOS when exporting the problem or configuring the solver.
- `'primals'` – A dictionary mapping PICOS variables to their value in the solution produced by the solver.
- `'duals'` – A list of dual values produced by the solver, in the order in which the constraints were added.

Warning: Any supplied options will be stored in the problem as if they were set via *set_option*.

Note: If the problem is dualized or cast as a SOCP during solution search, then it will be solved from scratch upon subsequent searches, even if the solver supports problem updates efficiently.

update_options(***options*)
Sets multiple solver options at once.

Parameters **options** – A parameter sequence of the form `key = value`.

For a list of available options and their default values, see the documentation of *set_option*.

verbosity()

Returns The problem's current verbosity level.

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

Parameters

- **filename** (*str*) – Path and name of the output file. The export format is inferred from the file extension. Supported extensions and their associated format are:
  - `'.cbf'` – Conic Benchmark Format.

    This format is suitable for optimization problems involving second order and/or semidefinite cone constraints. This is a standard choice for conic optimization problems. Visit the website of The Conic Benchmark Library or read A benchmark library for conic mixed-integer and continuous optimization by Henrik A. Friberg for more information.
  - `'.lp'` – LP format.

    This format handles only linear constraints, unless the writer `'cplex'` is used. In the latter case the extended CPLEX LP format is used instead.
  - `'.mps'` – MPS format.

    As the writer, you need to choose one of `'cplex'`, `'gurobi'` or `'mosek'`.
  - `'.opf'` – OPF format.

    As the writer, you need to choose `'mosek'`.
  - `'.dat-s'` – Sparse SDPA format.

    This format is suitable for semidefinite programs. Second order cone constraints are stored as semidefinite constraints on an *arrow shaped* matrix.
- **writer** (*str*) – The default `'picos'` denotes PICOS' internal writer, which can export to *LP*, *CBF*, and *Sparse SDPA* formats. If CPLEX, Gurobi or MOSEK is installed, you can choose `'cplex'`, `'gurobi'`, or `'mosek'`, respectively, to make use of that solver's export function and get access to more formats.

Warning: For problems involving a symmetric matrix variable X (typically, semidefinite programs), the expressions involving X are stored in PICOS as a function of $svec(X)$, the symmetric vectorized form of X (see Dattorro, ch.2.2.2.1), and are also exported in that form. As a result, using an external solver on a problem description file exported by PICOS will also yield a solution in this symmetric vectorized form.

The CBF writer tries to write symmetric variables X in the section `PSDVAR` of the .cbf file. However, this is possible only if the constraint $X \succeq 0$ appears in the problem, and no other LMI involves X. If these two conditions are not satisfied, then the symmetric vectorization of X is used as a (free) variable of the section `VAR` in the .cbf file, as explained in the previous paragraph.

QuadAsSocpError

exception picos.**QuadAsSocpError**(*msg*)

Bases: `Exception`

Exception raised when the problem can not be solved in the current form, because quad constraints are not handled. User should try to convert the quads as socp.

new_problem

picos.**new_problem**

alias of *picos.problem.Problem*

Variables

LOCATION	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
VERSION_FILE	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

LOCATION

picos.**LOCATION = '/builds/picos-api/picos/picos'**
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

VERSION_FILE

picos.**VERSION_FILE = '/builds/picos-api/picos/picos/.version'**
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

2.4.2 picos.constraints

This package contains the constraint types that are used to express optimization constraints. You do not need to instanciate these constraints directly; it is more convenient to create them by applying Python's comparison operators to algebraic expressions (see the *Cheat Sheet*).

Classes

AbsoluteValueConstraint(signedScalar, upperBound)	
AffineConstraint(lhs, relation, rhs[, ...])	An equality or inequality between two affine expressions.
Constraint(typeTerm[, customString, printSize])	An abstract base class for optimization constraints.
DetRootNConstraint(detRootN, lowerBound)	
ExpConeConstraint(element[, customString])	An exponential cone constraint stating that (x, y, z) fulfills $x \geq ye^{\frac{z}{y}} \wedge x > 0 \wedge y > 0$.
FlowConstraint(G, f, source, sink, flow_value)	
GeoMeanConstraint(geoMean, lowerBound)	
KullbackLeiblerConstraint(divergence, upperBound)	
LMIConstraint(lhs, relation, rhs[, customString])	An inequality with respect to the positive semidefinite cone, also known as a Linear Matrix Inequality (LMI) or an SDP constraint.
LSEConstraint(lse, upperBound)	An upper bound on a log-sum-exp expression.
LogConstraint(log, lowerBound)	A lower bound on a logarithmic expression.
MetaConstraint(tmpProblem, typeTerm[, ...])	An abstract base class for optimization constraints that are comprised of auxiliary variables and constraints.
PNormConstraint(pNorm, relation, rhs)	
PQNormConstraint(pqNorm, upperBound)	
QuadConstraint(lowerEqualZero[, customString])	An upper bound on a scalar quadratic expression.

Continued on next page

Table 6 – continued from previous page

RSOCConstraint(normedExpression, . . . [, . . .])	A rotated second order cone constraint.
SOCConstraint(normedExpression, upperBound)	A second order cone (2-norm cone, Lorentz cone) constraint.
SumExpConstraint(theSum, upperBound)	
SumExtremesConstraint(theSum, relation, rhs)	
SymTruncSimplexConstraint(simplex, element)	
TracePowConstraint(power, relation, rhs)	

AbsoluteValueConstraint

class picos.constraints.**AbsoluteValueConstraint**(*signedScalar*, *upperBound*)

Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

AffineConstraint

class picos.constraints.**AffineConstraint**(*lhs*, *relation*, *rhs*, *customString=None*)
Bases: *picos.constraints.Constraint*

An equality or inequality between two affine expressions.

Attributes Summary

EQ	
GE	
LE	
dual	
ge0	The expression posed to be greater or equal than zero in case of an inequality, otherwise the right hand side minus the left hand side.
greater	The greater-or-equal side expression in case of an inequality, otherwise the right hand side.
le0	The expression posed to be less or equal than zero in case of an inequality, otherwise the left hand side minus the right hand side.
size	

Continued on next page

Table 9 – continued from previous page

slack	
smaller	The smaller-or-equal side expression in case of an inequality, otherwise the left hand side.

Methods Summary

bounded_linear_form()	Separates the constraint into a linear function on the left hand side and a constant bound on the right hand side.
constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	
sparse_Ab_rows(varOffsetMap[, indexFunction])	A sparse list representation of the constraint, given a mapping of PICOS variables to column offsets (or alternatively given an index function).

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

ge0
The expression posed to be greater or equal than zero in case of an inequality, otherwise the right hand side minus the left hand side.

greater
The greater-or-equal side expression in case of an inequality, otherwise the right hand side.

le0
The expression posed to be less or equal than zero in case of an inequality, otherwise the left hand side minus the right hand side.

size

slack

smaller
The smaller-or-equal side expression in case of an inequality, otherwise the left hand side.

Methods Documentation

bounded_linear_form()
Separates the constraint into a linear function on the left hand side and a constant bound on the right hand side.

Returns A pair `(linear, bound)` where `linear` is a pure linear expression and `bound` is a constant expression.

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

sparse_Ab_rows(*varOffsetMap*, *indexFunction=None*)
A sparse list representation of the constraint, given a mapping of PICOS variables to column offsets (or alternatively given an index function).

The constraint is brought into a bounded linear form A • b, where • is one of $\leq$, $\geq$, or =, depending on the constraint relation, and the rows returned correspond to the matrix [A|b].

Parameters

- **varOffsetMap** (*dict*) – Maps variables or variable start indices to column offsets.
- **indexFunction** – Instead of adding the local variable index to the value returned by varOffsetMap, use the return value of this function, that takes as argument the variable and its local index, as the "column index", which need not be an integer. When this parameter is passed, the parameter varOffsetMap is ignored.

Returns A list of triples (J, V, c) where J contains column indices (representing scalar variables), V contains coefficients for each column index and c is a constant term. Each entry of the list represents a row in a constraint matrix.

Constraint

class picos.constraints.**Constraint**(*typeTerm*, *customString=None*, *printSize=False*)
Bases: abc.ABC

An abstract base class for optimization constraints.

Implementations

- need to implement the abstract methods `_str`, `_expression_names`, `_get_size`, `_get_slack` and `_set_dual`,

- need to oberwrite `_variable_names`, if applicable, and
- are supposed to call *Constraint.__init__* from within their own implementation of *__init__*.

Attributes Summary

EQ
GE
LE
dual
size
slack

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

size

slack

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

DetRootNConstraint

class picos.constraints.**DetRootNConstraint**(*detRootN*, *lowerBound*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

ExpConeConstraint

class picos.constraints.**ExpConeConstraint**(*element*, *customString=None*)
Bases: *picos.constraints.Constraint*

An exponential cone constraint stating that (x, y, z) fulfills $x \geq ye^{\frac{z}{y}} \wedge x > 0 \wedge y > 0$.

Attributes Summary

EQ
GE
LE
dual
size
slack
x
y
z

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

size

slack

x

y

z

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

FlowConstraint

class picos.constraints.**FlowConstraint**(*G*, *f*, *source*, *sink*, *flow_value*, *capacity=None*, *graphName=''*)

Bases: *picos.constraints.MetaConstraint*

Note: Unlike other *MetaConstraint* implementations, this one is used (via a wrapper function) by the user, so it is raising exceptions instead of making assertions if it is instanciated incorrectly.

Attributes Summary

EQ	
GE	
LE	
constraints	
dual	
prefix	
size	
slack	
variableNames	
variables	

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
draw()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

draw()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

GeoMeanConstraint

class picos.constraints.**GeoMeanConstraint**(*geoMean*, *lowerBound*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

KullbackLeiblerConstraint

class picos.constraints.**KullbackLeiblerConstraint**(*divergence*, *upperBound*)

Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
denominator
dual
factor
numerator
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

denominator

dual

factor

numerator

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

LMIConstraint

class picos.constraints.**LMIConstraint**(*lhs*, *relation*, *rhs*, *customString=None*)
Bases: *picos.constraints.Constraint*

An inequality with respect to the positive semidefinite cone, also known as a Linear Matrix Inequality (LMI) or an SDP constraint.

Attributes Summary

EQ	
GE	
LE	
dual	
greater	The greater-or-equal side expression.
nnd	The matrix expression posed to be nonnegative definite.
npd	The matrix expression posed to be nonpositive definite.
nsd	The matrix expression posed to be negative semidefinite.

Continued on next page

Table 23 – continued from previous page

psd	The matrix expression posed to be positive semidefinite.
size	
slack	
smaller	The smaller-or-equal side expression.

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

greater
The greater-or-equal side expression.

nnd
The matrix expression posed to be nonnegative definite.

npd
The matrix expression posed to be nonpositive definite.

nsd
The matrix expression posed to be negative semidefinite.

psd
The matrix expression posed to be positive semidefinite.

size

slack

smaller
The smaller-or-equal side expression.

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

LSEConstraint

class picos.constraints.**LSEConstraint**(*lse*, *upperBound*)
Bases: *picos.constraints.MetaConstraint*

An upper bound on a log-sum-exp expression.

Attributes Summary

EQ	
GE	
LE	
constraints	
dual	
exponents	The exponents of the logarithm of sum of exponentials expression.
le0	The logarithm of sum of exponentials expression after the constraint was reformulated to have an upper bound of zero.
le0Exponents	The exponents of the logarithm of sum of exponentials expression after the constraint was reformulated to have an upper bound of zero.
prefix	
size	
slack	
variableNames	
variables	

Methods Summary

constring()	

Continued on next page

Table 26 – continued from previous page

copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
has_zero_bound()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

exponents
The exponents of the logarithm of sum of exponentials expression.

le0
The logarithm of sum of exponentials expression after the constraint was reformulated to have an upper bound of zero.

le0Exponents
The exponents of the logarithm of sum of exponentials expression after the constraint was reformulated to have an upper bound of zero.

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

has_zero_bound()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

LogConstraint

class picos.constraints.**LogConstraint**(*log*, *lowerBound*)
Bases: *picos.constraints.MetaConstraint*

A lower bound on a logarithmic expression.

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

MetaConstraint

class picos.constraints.**MetaConstraint**(*tmpProblem*, *typeTerm*, *customString=None*)
Bases: *picos.constraints.Constraint*

An abstract base class for optimization constraints that are comprised of auxiliary variables and constraints.

Implementations

- need to implement the abstract method `_get_prefix`,
- need to implement *Constraint*'s abstract methods `_str` and `_get_slack`,
- may overwrite the default implementation for *Constraint*'s abstract methods `_get_size` and `_get_dual`, and
- are supposed to receive or construct a temporary problem containing the auxiliary objects and pass it to *MetaConstraint.__init__* (along with a number of standard parameters that are dispatched to *Constraint.__init__*) from within their own implementation of *__init__*.

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

PNormConstraint

class picos.constraints.**PNormConstraint**(*pNorm*, *relation*, *rhs*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
isGeneralized()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.

Continued on next page

Table 32 – continued from previous page

is_meta()
is_real()
keyconstring()

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

isGeneralized()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

PQNormConstraint

class picos.constraints.**PQNormConstraint**(*pqNorm*, *upperBound*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE

Continued on next page

Table 33 – continued from previous page

LE	
constraints	
dual	
prefix	
size	
slack	
variableNames	
variables	

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

QuadConstraint

class picos.constraints.**QuadConstraint**(*lowerEqualZero*, *customString=None*)
Bases: *picos.constraints.Constraint*

An upper bound on a scalar quadratic expression.

Attributes Summary

EQ
GE
LE
dual
size
slack

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

size

slack

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

RSOCConstraint

class picos.constraints.**RSOCConstraint**(*normedExpression*, *upperBoundFactor1*, *upperBoundFactor2=None*, *customString=None*)

Bases: *picos.constraints.Constraint*

A rotated second order cone constraint.

Attributes Summary

EQ	
GE	
LE	
dual	
size	
slack	

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	

Continued on next page

Table 38 – continued from previous page

is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

size

slack

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

SOCConstraint

class picos.constraints.**SOCConstraint**(*normedExpression*, *upperBound*, *customString=None*)
Bases: *picos.constraints.Constraint*

A second order cone (2-norm cone, Lorentz cone) constraint.

Attributes Summary

EQ	
GE	
LE	
dual	
size	
slack	

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	Deletes the constraint from the problem it is assigned to.
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

dual

size

slack

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()
Deletes the constraint from the problem it is assigned to.

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

SumExpConstraint

class picos.constraints.**SumExpConstraint**(*theSum*, *upperBound*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
denominator
dual
factor
numerator
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

denominator

dual

factor

numerator

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

SumExtremesConstraint

class picos.constraints.**SumExtremesConstraint**(*theSum*, *relation*, *rhs*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack

Continued on next page

Table 43 – continued from previous page

variableNames	
variables	

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

SymTruncSimplexConstraint

class picos.constraints.**SymTruncSimplexConstraint**(*simplex*, *element*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
prefix
size
slack
variableNames
variables

Methods Summary

constring()	
copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

TracePowConstraint

class picos.constraints.**TracePowConstraint**(*power*, *relation*, *rhs*)
Bases: *picos.constraints.MetaConstraint*

Attributes Summary

EQ
GE
LE
constraints
dual
lhs
prefix
size
slack
variableNames
variables

Methods Summary

constring()

Continued on next page

Table 48 – continued from previous page

copy_with_new_vars(newVars[, newCons])	
delete()	
expressions()	
isTrace()	
is_complex()	
is_decreasing()	Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.
is_equality()	Whether the constraints states the equality between the left hand side and the right hand side.
is_increasing()	Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.
is_inequality()	Whether the constraints states an inequality between the left hand side and the right hand side.
is_meta()	
is_real()	
keyconstring()	

Attributes Documentation

EQ = '='

GE = '>'

LE = '<'

constraints

dual

lhs

prefix

size

slack

variableNames

variables

Methods Documentation

constring()

copy_with_new_vars(*newVars*, *newCons=None*)

delete()

expressions()

isTrace()

is_complex()

is_decreasing()
Whether the constraint states exactly that the left hand side is greater or equal than the right hand side.

is_equality()
Whether the constraints states the equality between the left hand side and the right hand side.

is_increasing()
Whether the constraint states exactly that the left hand side is smaller or equal than the right hand side.

is_inequality()
Whether the constraints states an inequality between the left hand side and the right hand side.

is_meta()

is_real()

keyconstring()

2.4.3 picos.expressions

This module contains the expression types created by PICOS when you perform algebraic operations on variables and parameters. You do not need to create expressions directly; it is more convenient to use standard Python operators (see the *Cheat Sheet*) and additional algebraic functions (see the `picos` and `picos.tools` namespaces) on the basic affine expressions created by `add_variable` and `new_param`.

Members

class picos.expressions.**AffinExp**(*factors=None*, *constant=None*, *size=(1, 1)*, *string='0'*, *typeBaseStr='Affine Expression'*)
Bases: `picos.expressions.Expression`

A class for defining vectorial (or matrix) affine expressions.

Overloaded operators

+ sum (with an affine or quadratic expression)

+= in-place sum (with an affine or quadratic expression)

− unitary minus or substraction (of an affine or quadratic expression) or unitary

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

^ hadamard product (elementwise multiplication with another affine expression, similarly as MATLAB operator `.*`)

/ 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 any nonzero exponent otherwise). In the case of a nonconstant affine expression, the exponentiation returns a quadratic expression if the exponent is 2, or a `TracePow_Exp` object for other exponents. A rational approximation of the exponent is used, and the power inequality is internally replaced by an equivalent set of second order cone inequalities.

& 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 $\preceq$); or, if the right hand side is a `Set`, membership in this set.

>> 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:

- The rotated second order cone constraint `abs(exp1)**2 < exp2 * exp3` implicitely assumes that the scalar expression `exp2` (and hence `exp3`) **is nonnegative**.
- Inequalities involving an exponentiation of the form `x**p` where `p` is not an even positive integer impose nonnegativity on `x`.
- 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.

Htranspose()
Hermitian (or conjugate) transposition.

apply_function(*fun*)

conjugate()
Complex conjugate.

copy()

del_imag()

del_real()

del_type()

del_value()

diag(*dim*)

eval(*ind=None*)

classmethod fromMatrix(*matrix*, *size=None*)

classmethod fromScalar(*scalar*)

get_imag()

get_real()

get_type()

hadamard(*fact*)
hadamard (elementwise) product

inplace_Htranspose()

inplace_conjugate()

inplace_partial_transpose(*dim_1=None*, *subsystems=None*, *dim_2=None*)

inplace_transpose()

is0()

Returns Whether this is a scalar, vector or matrix of all zeros.

is1()

Returns Whether this is a scalar or vector of all ones.

is_pure_antisym_var()

is_pure_complex_var()

is_real()

is_valued(*ind=None*)

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

partial_trace(*k=1*, *dim=None*)
Partial trace, see also *the partial_trace tool*

partial_transpose(*dim_1=None*, *subsystems=None*, *dim_2=None*)
Partial transposition.

same_as(*other*)

set_imag(*value*)

set_real(*value*)

set_type(*value*)

set_value(*value*)

soft_copy()

sparse_rows(*varOffsetMap*, *lowerTriangle=False*, *upperTriangle=False*, *indexFunction=None*)
A sparse list representation of the expression, given a mapping of PICOS variables to column offsets (or alternatively given an index function).

Parameters

- **varOffsetMap** (*dict*) – Maps variables or variable start indices to column offsets.
- **lowerTriangle** (*bool*) – Whether to return only the lower triangular part of the expression.
- **lowerTriangle** – Whether to return only the upper triangular part of the expression.
- **indexFunction** – Instead of adding the local variable index to the value returned by varOffsetMap, use the return value of this function, that takes as argument the variable and its local index, as the "column index", which need not be an integer. When this parameter is passed, the parameter varOffsetMap is ignored.

Returns A list of triples (J, V, c) where J contains column indices (representing scalar variables), V contains coefficients for each column index and c is a constant term. Each entry of the list represents a row in the vectorization of the expression.

transpose()
Matrix transposition.

classmethod zero(*size=(1, 1)*)

H
Hermitian (or conjugate) transposition.

T
Matrix transposition.

Tx
Partial transposition for an $n^2 \times n^2$ matrix, assuming subblocks of size n×n. Refer to *partial_transpose*.

conj
Complex conjugate.

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*.

imag
imaginary part (for complex expressions)

real
real part (for complex expressions)

size
Size of the affine expression.

typeStr

vtype

class picos.expressions.**Ball**(*p*, *radius*)
Bases: *picos.expressions.Set*

Represents a ball of a given norm.

** Overloaded operators **

>> membership of the right hand side in this set.

class picos.expressions.**DetRootN_Exp**(*exp*)
Bases: *picos.expressions.Expression*

A class storing the n-th root of the determinant of a positive semidefinite matrix.

Use the function *picos.detrootn* to create an instance of this class.

Note that the matrix X is forced to be positive semidefinite when a constraint of the form `t < pic.detrootn(X)` is added.

Overloaded operator

> greater **or equal** than (a scalar affine expression)

eval(*ind=None*)

dim = None
dimension of `exp`

exp = None
The affine expression to which the det-root-n is applied

class picos.expressions.**ExponentialCone**
Bases: *picos.expressions.Set*

Represents the cone $\text{closure}\{(x, y, z) : y \exp(z/y) \leq x, x, y > 0\}$.

** Overloaded operators **

>> membership of the right hand side in this set.

class picos.expressions.**Expression**(*typeStr*, *symbStr*)
Bases: `object`

The parent class of all expressions, including variables.

del_value()

eval()

get_value()

get_value_as_matrix()

has_complex_coef()

is_valued()

Returns Whether the expression is valued.

Example

```
>>> import picos as pic
>>> prob=pic.Problem()
>>> x=prob.add_variable('x',2)
>>> x.is_valued()
False
>>> print(abs(x))
‖x‖
>>> x.value=[3,4]
>>> abs(x).is_valued()
True
>>> print(abs(x))
5.0
```

set_value(*value*)

string = None
A symbolic string representation of the expression. It is always used by __descr__, and it is equivalent to the value returned by __str__ when the expression is not fully valued.

typeStr = None
A string describing the expression type.

value
Value of the expression.

It is defined (not None) in the following three situations:

- The expression is constant.
- Every variable involved in the expression is valued (this can be done by setting `value` on each of the variables).
- The expression involves variables of a problem that has already been solved, so that their values are set to some optimum value.

Unlike *valueAsMatrix*, scalars are returned as scalar types.

valueAsMatrix
Value of the expression.

Refer to *value* for when it is available.

Unlike *value*, scalars are returned in the form of 1x1 matrices.

class picos.expressions.**GeneralFun**(*fun*, *Exp*, *funstring*)
Bases: *picos.expressions.Expression*

A class storing a scalar function, applied to an affine expression.

eval(*ind=None*)

Exp = None
The affine expression to which the function is applied

fun = None
The function `f` applied to the affine expression. This function must take in argument a `cvxopt sparse matrix` `X`. Moreover, the call `fx,grad,hess=f(X)` must return the function value $f(X)$ in `fx`, the gradient $\nabla f(X)$ in the `cvxopt matrix` `grad`, and the Hessian $\nabla^2 f(X)$ in the `cvxopt sparse matrix` `hess`.

funstring = None
a string representation of the function name

class picos.expressions.**GeoMeanExp**(*exp*)
Bases: *picos.expressions.Expression*

A class storing the geometric mean of a multidimensional expression.

Overloaded operator

> **>** greater **or equal** than (the rhs must be a scalar affine expression)

eval(*ind=None*)

exp = None
The affine expression to which the geomean is applied

class picos.expressions.**KullbackLeibler**(*exp*, *exp2=None*)
Bases: *picos.expressions.Expression*

A Kullback-Leibler divergence.

If an affine expression x of size N with elements $x_1, x_2, \ldots, x_N$ is given, then *KullbackLeibler(x)* represents the (negative) entropy $\sum_{i=1}^{N} x_i \log(x_i)$.

If a second expression y (of same dimension as x) is given, then *KullbackLeibler(x, y)* represents $\sum_{i=1}^{N} x_i \log x_i/y_i$.

Overloaded operators

> **+** addition (with a scalar *AffinExp* or another *KullbackLeibler*)
>
> **<** less **or equal** (than a scalar *AffinExp*)

Note: Upper-bounding a Kullback-Leibler divergence adds the implicit constraints $x > 0$ and $y > 0$.

---

eval(*ind=None*)

class picos.expressions.**LogSumExp**(*exp*)
Bases: *picos.expressions.Expression*

The logarithm of a sum of exponentials.

If an affine expression x of size N with elements $x_1, x_2, \ldots, x_N$ is given, then *LogSumExp(x)* represents the expression $\log \sum_{i=1}^{N} e^{x_i}$.

Overloaded operators

> **<** less **or equal** than

eval(*ind=None*)

class picos.expressions.**Logarithm**(*exp*)
Bases: *picos.expressions.Expression*

The logarithm of a nonconstant scalar affine expression.

Overloaded operators

> **+** addition (with a scalar *AffinExp*)
>
> ***** multiplication (with the expression under the logarithm)
>
> **>** greater **or equal** (than a scalar *AffinExp*)

Note: Lower-bounding $\log(x)$ adds the implicit constraint $x \geq 0$.

---

eval(*ind=None*)

class picos.expressions.**Norm**(*exp*)
Bases: *picos.expressions.Expression*

Euclidean (or Frobenius) norm of an Affine Expression.

Overloaded operators

> ****** exponentiation (with an exponent of 2)

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

eval(*ind=None*)

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

class picos.expressions.**NormP_Exp**(*exp*, *numerator*, *denominator=1*, *num2=None*, *den2=1*)
Bases: *picos.expressions.Expression*

A class storing the entrywise p-norm of a multidimensional expression.

Use the function *picos.norm()* to create an instance of this class. This class can also be used to store the $L_{p,q}$ norm of a matrix.

Generalized norms are also defined for $p < 1$, by using the usual formula $\mathrm{norm}(\mathbf{x}, p) := \left(\sum_i x_i^p\right)^{1/p}$. Note that this function is concave (for $p < 1$) over the set of vectors with nonnegative coordinates. When a constraint of the form $\mathrm{norm}(\mathbf{x}, p) > t$ with $p \leq 1$ is entered, PICOS implicitely forces $\mathbf{x}$ to be a nonnegative vector.

Overloaded operator

< less **or equal** than (a scalar affine expression, p >= 1)

> greater **or equal** than (a scalar affine expression, p <= 1)

eval(*ind=None*)

den2 = None
denominator of q

denominator = None
denominator of p

exp = None
The affine expression to which the p-norm is applied

num2 = None
numerator of q

numerator = None
numerator of p

class picos.expressions.**QuadExp**(*quad*, *aff*, *string*, *LR=None*)
Bases: *picos.expressions.Expression*

Quadratic expression.

Overloaded operators

+ addition (with an affine or a quadratic expression)

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

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

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

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

copy()

eval(*ind=None*)

nnz()

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 $\|\texttt{aff}\|^2$, and
- `LR=(aff1,aff2)` when the expression is equal to `aff1*aff2`.

aff = None
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*.

class picos.expressions.**Set**(*typeStr*, *symbStr=None*)
Bases: object

Parent class for set expressions.

string

class picos.expressions.**SumExponential**(*exp*, *exp2=None*)
Bases: *picos.expressions.Expression*

A sum of (perspectives of) exponentials.

If an affine expression x of size N with elements $x_1, x_2, \ldots, x_N$ is given, then *SumExponential(x)* represents the expression $\sum_{i=1}^{N} e^{x_i}$.

If a second affine expression y (of same dimension as x) is given, then *SumExponential(x, y)* represents $\sum_{i=1}^{N} y_i e^{x_i/y_i}$.

Overloaded operators

- **+** addition (with a scalar *AffinExp* or another *SumExponential*)
- ***** multiplication (with a scalar *AffinExp*)
- **/** division (by a scalar *AffinExp*)
- **<** less **or equal** (than a scalar *AffinExp* or another *SumExponential*)

Note: Upper-bounding with a nonconstant scalar t adds the implicit constraint $t > 0$. Upper-bounding a sum of perspectives of exponentials (that is in the case of $y \neq 1$ given) independently adds the implicit constraint $y > 0$.

---

eval(*ind=None*)

class picos.expressions.**Sum_k_Largest_Exp**(*exp*, *k*, *eigenvals=False*)
Bases: *picos.expressions.Expression*

A class storing the sum of the k largest elements of an *AffinExp*, or the sum of its k largest eigenvalues (for a square matrix expression).

Use the function *picos.sum_k_largest* or *picos.sum_k_largest_lambda* to create an instance of this class.

Note that the matrix X is assumed to be symmetric when a constraint of the form `pic.sum_k_largest_lambda(X,k) < t` is added.

Overloaded operator

- **<** smaller **or equal** than (a scalar affine expression)

eval(*ind=None*)

eigenvalues = None
whether this is a sum of k largest eigenvalues (for symmetric matrices)

exp = None
The affine expression to which the sum_k_largest is applied

k = None
The number of elements to sum

class picos.expressions.**Sum_k_Smallest_Exp**(*exp*, *k*, *eigenvals=False*)
Bases: *picos.expressions.Expression*

A class storing the sum of the k smallest elements of an *AffinExp*, or the sum of its k smallest eigenvalues (for a square matrix expression).

Use the function *picos.sum_k_smallest()* or *picos.sum_k_smallest_lambda()* to create an instance of this class.

Note that the matrix X is assumed to be symmetric when a constraint of the form `pic.sum_k_smallest_lambda(X,k) > t` is added.

Overloaded operator

> greater **or equal** than (a scalar affine expression)

eval(*ind=None*)

eigenvalues = None
whether this is a sum of k smallest eigenvalues (for symmetric matrices)

exp = None
The affine expression to which sum_k_smallest is applied

k = None
The number of elements to sum

class picos.expressions.**TracePow_Exp**(*exp*, *numerator*, *denominator=1*, *M=None*)
Bases: *picos.expressions.Expression*

The p-th power of a scalar, or more generally the trace of the p-th power of a symmetric matrix, for some rational p.

The exponent p is given in the form of a numerator/denominator pair.

You can use the shorthand function *picos.tracepow()* and the overloaded exponentiation operator `**` to create an instance of this class.

Note that this expression is concave for $0 \leq p \leq 1$ and convex for both $p \leq 0$ and *:math:p geq 1* for a non-negative (positive semidefinite) base. If the expression is concave, then an additional positive semdidefinite coefficient matrix M may be given so that the expression describes $\mathrm{trace}(MX^p)$.

Warning: The symmetry of the base matrix and the positive semidefiniteness of the optional coefficient matrix are not checked or enforced by PICOS.

Overloaded operator

< less **or equal** than (a scalar affine expression, for a convex power)

> greater **or equal** than (a scalar affine expression, for a concave power)

eval(*ind=None*)

M = None
the coef matrix

denominator = None
denominator of p

dim = None
dimension of `exp`

exp = None
The affine expression to which the p-norm is applied

numerator = None
numerator of p

class picos.expressions.**TruncatedSimplex**(*radius=1*, *truncated=False*, *nonneg=True*)
Bases: *picos.expressions.Set*

Represents a simplex, that can be intersected with the ball of radius 1 for the infinity-norm (truncation), and that can be symmetrized with respect to the origin.

** Overloaded operators **

>> membership of the right hand side in this set.

class picos.expressions.**Variable**(*parent_problem*, *name*, *size*, *Id*, *startIndex*, *vtype='continuous'*, *lower=None*, *upper=None*)
Bases: *picos.expressions.AffinExp*

This class stores a variable.

bound_constraint()
Returns the variable bounds in the form of a PICOS multidimensional *affine constraint*.

del_value(*index=None*)

eval(*ind=None*)

set_lower(*lo*)
sets a lower bound to the variable (lo may be scalar or a matrix of the same size as the variable `self`). Entries smaller than -INFINITY = -1e16 are ignored

set_sparse_lower(*indices*, *bnds*)
sets the lower bound bnds[i] to the index indices[i] of the variable.

For a symmetric matrix variable, bounds on elements in the upper triangle are ignored.

Parameters

- **indices** (`list`) – list of indices, given as integers (column major order) or tuples (i,j).
- **bnds** – list of lower bounds.

Warning: This function does not modify the existing bounds on elements other than those specified in `indices`.

Example:

```
>>> import picos as pic
>>> P = pic.Problem()
>>> X = P.add_variable('X',(3,2),lower = 0)
>>> X.set_sparse_upper([0,(0,1),1],[1,2,0])
>>> X.bnd #doctest: +NORMALIZE_WHITESPACE
{0: (0.0, 1.0),
 1: (0.0, 0.0),
 2: (0.0, None),
 3: (0.0, 2.0),
 4: (0.0, None),
 5: (0.0, None)}
```

set_sparse_upper(*indices*, *bnds*)
sets the upper bound bnds[i] to the index indices[i] of the variable.

For a symmetric matrix variable, bounds on elements in the upper triangle are ignored.

Parameters

- **indices** (list) – list of indices, given as integers (column major order) or tuples (i,j).
- **bnds** – list of upper bounds.

Warning: This function does not modify the existing bounds on elements other than those specified in indices.

set_upper(*up*)
sets an upper bound to the variable (up may be scalar or a matrix of the same size as the variable self). Entries larger than INFINITY = 1e16 are ignored

set_value(*value*, *index=None*)

Id = None
An integer index (obsolete)

bnd
var.bnd[i] returns a tuple (lo,up) of lower and upper bounds for the i-th element of the variable var. None means +/- infinite. if var.bnd[i] is not defined, then var[i] is unbounded.

dim
The algebraic dimension of the variable.

endIndex
end position in the global vector of all variables

name = None
The name of the variable (str)

origin = None
The metaconstraint that created this variable, if any.

parent_problem = None
The Problem instance to which this variable belongs

semiDef = None
Conditionally evaluates to True if this is a symmetric variable subject to X >> 0. Counts the number of constraints stating this.

startIndex
starting position in the global vector of all variables

typeStr

vtype
one of the following strings:

- 'continuous' (continuous variable)
- 'binary' (binary 0/1 variable)
- 'integer' (integer variable)
- 'symmetric' (symmetric matrix variable)
- 'antisym' (antisymmetric matrix variable)
- 'complex' (complex matrix variable)
- 'hermitian' (complex hermitian 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])

2.4.4 picos.glyphs

PICOS internally uses this module to produce string representations for the algebraic expressions that you create. The function-like objects that are used to build such strings are called "glyphs" and are instanciated by this module following the singleton pattern. As a result, you can modify the glyph objects listed below to influence how PICOS will format future strings, for example to disable use of unicode symbols that your console does not suppport or to adapt PICOS' output to the rest of your application.

Here's an example of first swapping the entire character set to display expressions using only Latin-1 characters, and then modifying a single glyph to our liking:

```
>>> import picos
>>> X = picos.new_problem().add_variable("X", (2,2), "symmetric")
>>> print(X >> 0)
X ≽ 0
>>> picos.glyphs.latin1()
>>> print(X >> 0)
X » 0
>>> picos.glyphs.psdge.template = "{} - {} is psd"
>>> print(X >> 0)
X - 0 is psd
```

Note that glyphs understand some algebraic rules such as operator precedence and associativity. This is possible because strings produced by glyphs remember how they were created.

```
>>> one_plus_two = picos.glyphs.add(1, 2)
>>> one_plus_two
'1 + 2'
>>> one_plus_two.glyph.template, one_plus_two.operands
('{} + {}', (1, 2))
>>> picos.glyphs.add(one_plus_two, 3)
'1 + 2 + 3'
>>> picos.glyphs.sub(0, one_plus_two)
'0 - (1 + 2)'
```

The positive semidefinite glyph above does not yet know how to properly handle arguments with respect to the − symbol involved, but we can modify it further:

```
>>> print(X + X >> X + X)
X + X - X + X is psd
>>> # Use the same operator binding strength as regular substraction.
>>> picos.glyphs.psdge.order = picos.glyphs.sub.order
>>> print(X + X >> X + X)
X + X - (X + X) is psd
```

You can reset all glyphs to their initial state as follows:

```
>>> picos.glyphs.default()
```

Members

class picos.glyphs.**Am**(*glyph*)
Bases: *picos.glyphs.Op*

A math atom glyph.

class picos.glyphs.**Br**(*glyph*)
Bases: *picos.glyphs.Op*

A math operator glyph with enclosing brackets.

class picos.glyphs.**Fn**(*glyph*)
Bases: *picos.glyphs.Op*

A math operator glyph in function form.

class picos.glyphs.**Gl**(*glyph*)
Bases: object

The basic "glyph", a wrapper for a format string that contains special symbols for building (algebraic) expressions.

Sublcasses are supposed to extend formatting routines, going beyond of what Python string formatting is capabale of. In particular, glyphs can be used to craft unambiguous algebraic expressions with the minimum amount of parenthesis.

rebuild()
If the template was created using other glyphs, rebuild it.

Returns True if the template has changed.

reset()

update(*new*)

class picos.glyphs.**GlStr**(*string*, *glyph*, *operands*)
Bases: str

A string created from a glyph.

It has an additional *glyph* field pointing to the glyph that created it, and a *operands* field containing the values used to create it.

reglyphed()

Returns A rebuilt version of the string using current glyphs.

glyph = None
The glyph used to create the string.

operands = None
The operands used to create the string.

class picos.glyphs.**Op**(*glyph*, *order*, *assoc=False*, *closed=False*)
Bases: *picos.glyphs.Gl*

The basic math operator glyph.

Parameters

- **glyph** (*str*) – A string format template denoting the symbols to be used.
- **order** (*int*) – The operator's position in the binding strength hierarchy. Operators with lower numbers have precedence (bind more strongly).
- **assoc** (*bool*) – If this is True, then the operator is associative, so that parenthesis are always omitted around operands with an equal outer operator. Otherwise, (1) parenthesis are used around the right hand side operand of a binary operation of same binding strength and (2) around all operands of non-binary operations of same binding strength.
- **closed** (*bool*) – If True, the operator already encloses the operands in some sort of brackets, so that no additional parenthesis are needed. For glyphs where only some operands are enclosed, this can be a list.

reset()

update(*new*)

class picos.glyphs.**OpStr**(*string*, *glyph*, *operands*)
Bases: *picos.glyphs.GlStr*

A string created from a math operator glyph.

class picos.glyphs.**Rl**(*glyph*)
Bases: *picos.glyphs.Op*

A math relation glyph.

class picos.glyphs.**Tr**(*glyph*)
Bases: *picos.glyphs.Op*

A math glyph in superscript/trailer form.

picos.glyphs.**ascii**()
Let PICOS create future string representations using only ASCII characters.

picos.glyphs.**cleverAdd**(*left*, *right*)
A wrapper around *add* that resorts to *sub* if the second operand was created by *neg* or is a negative number (string). In both cases the second operand is adjusted accordingly.

Example

```
>>> from picos.glyphs import neg, add, cleverAdd, matrix
>>> add("x", neg("y"))
'x + -y'
>>> cleverAdd("x", neg("y"))
'x - y'
>>> add("X", matrix(neg("y")))
'X + [-y]'
>>> cleverAdd("X", matrix(neg("y")))
'X - [y]'
>>> cleverAdd("X", matrix(-1.5))
'X - [1.5]'
```

picos.glyphs.**cleverNeg**(*value*)
A wrapper around *neg* that resorts to unnegating an already negated value.

Example

```
>>> from picos.glyphs import neg, cleverNeg, matrix
>>> neg("x")
'-x'
>>> neg(neg("x"))
'-(-x)'
>>> cleverNeg(neg("x"))
'x'
>>> neg(matrix(-1))
'-[-1]'
>>> cleverNeg(matrix(-1))
'[1]'
```

picos.glyphs.**cleverSub**(*left*, *right*)
A wrapper around *sub* that resorts to *add* if the second operand was created by *neg* or is a negative number(string). In both cases the second operand is adjusted accordingly.

Example

```
>>> from picos.glyphs import neg, sub, cleverSub, matrix
>>> sub("x", neg("y"))
'x - -y'
>>> cleverSub("x", neg("y"))
'x + y'
>>> sub("X", matrix(neg("y")))
'X - [-y]'
>>> cleverSub("X", matrix(neg("y")))
'X + [y]'
>>> cleverSub("X", matrix(-1.5))
'X + [1.5]'
```

picos.glyphs.**colVectorize**(**entries*)

picos.glyphs.**default**(*rebuildDerivedGlyphs=True*)
Let PICOS create future string representations using unicode characters.

picos.glyphs.**is_negated**(*value*)
Checks if a value can be unnegated by *unnegate*.

picos.glyphs.**latin1**(*rebuildDerivedGlyphs=True*)
Let PICOS create future string representations using ISO 8859-1 characters.

picos.glyphs.**makeFunction**(**names*)
Creates an ad-hoc composite function glyphs.

Example

```
>>> from picos.glyphs import makeFunction
>>> makeFunction("log", "sum", "exp")("x")
'logosumoexp(x)'
```

picos.glyphs.**matrixCat**(*left*, *right*, *horizontal=True*)
A clever wrapper around *matrix*, *horicat* and *vertcat*.

Example

```
>>> from picos.glyphs import matrixCat
>>> Z = matrixCat("X", "Y")
>>> Z
'[X, Y]'
>>> matrixCat(Z, Z)
'[X, Y, X, Y]'
>>> matrixCat(Z, Z, horizontal = False)
'[X, Y; X, Y]'
```

picos.glyphs.**rebuild**()
Updates glyphs that are based upon other glyphs.

picos.glyphs.**rowVectorize**(**entries*)

picos.glyphs.**scalar**(*value*)
This function mimics an operator glyph, but it returns a normal string (as opposed to an *OpStr*).

This is not realized as an atomic operator glyph to not increase the recursion depth of *is_negated* and *unnegate* unnecessarily.

Example

```
>>> from picos.glyphs import scalar
>>> str(1.0)
'1.0'
>>> scalar(1.0)
'1'
```

picos.glyphs.**show**(**args*)

picos.glyphs.**unicode**(*rebuildDerivedGlyphs=True*)
Let PICOS create future string representations using unicode characters.

picos.glyphs.**unnegate**(*value*)
Unnegates a value in a sensible way, more precisely by recursing through a sequence of glyphs used to create the value and for which we can factor out negation, and negating the underlaying (numeric or string) value.

Raises **ValueError** – When `is_negated(value)` returns `False`.

picos.glyphs.**Diag = <picos.glyphs.Fn object>**
Diagonal matrix glyph.

picos.glyphs.**abs = <picos.glyphs.Br object>**
Absolute value glyph.

picos.glyphs.**add = <picos.glyphs.Op object>**
Addition glyph.

picos.glyphs.**closure = <picos.glyphs.Fn object>**
Set closure glyph.

picos.glyphs.**compose = <picos.glyphs.Gl object>**
Function composition glyph.

picos.glyphs.**conj = <picos.glyphs.Tr object>**
Complex conugate glyph.

picos.glyphs.**cubed = <picos.glyphs.Tr object>**
Cubed value glyph.

picos.glyphs.**det = <picos.glyphs.Fn object>**
Determinant glyph.

picos.glyphs.**diag = <picos.glyphs.Fn object>**
Diagonal vector glyph.

picos.glyphs.**div = <picos.glyphs.Op object>**
Division glyph.

picos.glyphs.**dotp = <picos.glyphs.Br object>**
Scalar product glyph.

picos.glyphs.**element = <picos.glyphs.Rl object>**
Set element glyph.

picos.glyphs.**eq = <picos.glyphs.Rl object>**
Equality glyph.

picos.glyphs.**exp = <picos.glyphs.Fn object>**
Exponentiation glyph.

picos.glyphs.**forall = <picos.glyphs.Gl object>**
Universal quantification glyph.

picos.glyphs.**fromto = <picos.glyphs.Gl object>**
Range glyph.

picos.glyphs.**ge = <picos.glyphs.Rl object>**
Greater or equal glyph.

picos.glyphs.**gt = <picos.glyphs.Rl object>**
Greater than glyph.

picos.glyphs.**hadamard = <picos.glyphs.Op object>**
Hadamard product glyph.

picos.glyphs.**horicat = <picos.glyphs.Op object>**
Horizontal concatenation glyph.

picos.glyphs.**htransp = <picos.glyphs.Tr object>**
Matrix hermitian transposition glyph.

picos.glyphs.**idmatrix = <picos.glyphs.Am object>**
Identity matrix glyph.

picos.glyphs.**interval = <picos.glyphs.Gl object>**
Interval glyph.

picos.glyphs.**intrange = <picos.glyphs.Gl object>**
Integer range glyph.

picos.glyphs.**kron = <picos.glyphs.Op object>**
Kronecker product glyph.

picos.glyphs.**lambda_ = <picos.glyphs.Am object>**
Lambda symbol glyph.

picos.glyphs.**le = <picos.glyphs.Rl object>**
Lesser or equal glyph.

picos.glyphs.**log = <picos.glyphs.Fn object>**
Logarithm glyph.

picos.glyphs.**lt = <picos.glyphs.Rl object>**
Lesser than glyph.

picos.glyphs.**matrix = <picos.glyphs.Br object>**
Matrix glyph.

picos.glyphs.**max = <picos.glyphs.Fn object>**
Maximum glyph.

picos.glyphs.**min = <picos.glyphs.Fn object>**
Minimum glyph.

picos.glyphs.**mul = <picos.glyphs.Op object>**
Multiplication glyph.

picos.glyphs.**neg = <picos.glyphs.Op object>**
Negation glyph.

picos.glyphs.**norm = <picos.glyphs.Br object>**
Norm glyph.

picos.glyphs.**parenth = <picos.glyphs.Gl object>**
Parenthesis glyph.

picos.glyphs.**pnorm = <picos.glyphs.Op object>**
p-Norm glyph.

picos.glyphs.**power = <picos.glyphs.Tr object>**
Power glyph.

picos.glyphs.**pqnorm = <picos.glyphs.Op object>**
pq-Norm glyph.

picos.glyphs.**psdge = <picos.glyphs.Rl object>**
Lesser or equal w.r.t. the p.s.d. cone glyph.

picos.glyphs.**psdle = <picos.glyphs.Rl object>**
Greater or equal w.r.t. the p.s.d. cone glyph.

picos.glyphs.**ptrace = <picos.glyphs.Op object>**
Matrix p-Trace glyph.

picos.glyphs.**ptransp = <picos.glyphs.Tr object>**
Matrix partial transposition glyph.

picos.glyphs.**repr1 = <picos.glyphs.Gl object>**
Representation glyph.

picos.glyphs.**repr2 = <picos.glyphs.Gl object>**
Long representation glyph.

picos.glyphs.**sep = <picos.glyphs.Gl object>**
Seperator glyph.

picos.glyphs.**set = <picos.glyphs.Gl object>**
Set glyph.

picos.glyphs.**size = <picos.glyphs.Gl object>**
Matrix size/shape glyph.

picos.glyphs.**slice = <picos.glyphs.Op object>**
Expression slicing glyph.

picos.glyphs.**squared = <picos.glyphs.Tr object>**
Squared value glyph.

picos.glyphs.**sub = <picos.glyphs.Op object>**
Substraction glyph.

picos.glyphs.**sum = <picos.glyphs.Fn object>**
Summation glyph.

picos.glyphs.**trace = <picos.glyphs.Fn object>**
Matrix trace glyph.

picos.glyphs.**transp = <picos.glyphs.Tr object>**
Matrix transposition glyph.

picos.glyphs.**vertcat = <picos.glyphs.Op object>**
Vertical concatenation glyph.

2.4.5 picos.problem

The *Problem* class represents your optimization problem and is your main way of interfacing PICOS. After you create a problem instance, you can add variables to it via *add_variable* and use standard Python algebraic operators (*Cheat Sheet*) and algebraic functions (*picos.tools*) to create expressions and constraints involving these variables.

Members

class picos.problem.**Problem**(***options*)
Bases: object

PICOS' representation of an optimization problem.

Example

```
>>> import picos
>>> P = picos.Problem(verbose = 0)
>>> X = P.add_variable("X", (2,2), lower = 0)
>>> # 1|X is the dot prouct of X with a matrix of all ones.
>>> C = P.add_constraint(1|X < 10)
>>> P.set_objective("max", picos.trace(X))
>>> # PICOS will select a suitable solver if you don't specify one.
>>> solution = P.solve(solver = "cvxopt")
>>> solution["status"]
'optimal'
>>> solution["time"] #doctest: +SKIP
0.00034999847412109375
>>> round(P.obj_value(), 1)
10.0
>>> print(X) #doctest: +SKIP
[ 0.00e+00  0.00e+00]
[ 0.00e+00  1.00e+01]
>>> round(C.dual, 1)
1.0
```

Creates an empty problem and optionally sets initial solver options.

Parameters **options** – A parameter sequence of solver options.

add_constraint(*constraint*, *key=None*)
Adds a constraint to the problem.

Parameters

- **constraint** (*Constraint*) – The constraint to be added.
- **key** (*str*) – Optional name of the constraint.

Returns The constraint that was added to the problem, which may be a *MetaConstraint* that contains further references to auxiliary constraints that were also added, as well as potentially references to new auxiliary variables.

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

Adds a list of constraints to the problem, enabling the use of Python list comprehensions (see the example below).

Parameters

- **lst** (*list*) – A list of *constraints*.
- **it** – DEPRECATED
- **indices** – DEPRECATED
- **key** (*str*) – A name describing the list of constraints.

Returns A list of all constraints that were added.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> pprint(x)
[<2×1 Continuous Variable: x[0]>,
 <2×1 Continuous Variable: x[1]>,
 <2×1 Continuous Variable: x[2]>,
 <2×1 Continuous Variable: x[3]>,
 <2×1 Continuous Variable: x[4]>]
>>> y=prob.add_variable('y',5)
>>> IJ=[(1,2),(2,0),(4,2)]
>>> w={}
>>> for ij in IJ:
...         w[ij]=prob.add_variable('w[{},{}]'.format(*ij),3)
...
>>> u=pic.new_param('u',cvx.matrix([2,5]))
>>> C1=prob.add_list_of_constraints([u.T*x[i] < y[i] for i in range(5)])
>>> C2=prob.add_list_of_constraints([abs(w[i,j])<y[j] for (i,j) in IJ])
>>> C3=prob.add_list_of_constraints([y[t] > y[t+1] for t in range(4)])
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (SOCP):
24 variables, 9 affine constraints, 12 vars in 3 SO cones
<BLANKLINE>
w   : dict of 3 variables, (3, 1), continuous
x   : list of 5 variables, (2, 1), continuous
y   : (5, 1), continuous
<BLANKLINE>
    find vars
such that
  u|T·x[i] ≤ y[i] ∀ i ∈ [0...4]
  ‖w[i,j]‖ ≤ y[j] ∀ (i,j) ∈ zip([1,2,4],[2,0,2])
  y[i] ≥ y[i+1] ∀ i ∈ [0...3]
---------------------
```

add_variable (*name*, *size=1*, *vtype='continuous'*, *lower=None*, *upper=None*)

Adds a variable to the problem and returns it for use in constraints.

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 an n-dimensional vector,
  - or a `tuple` (n, m), in which case the variable is a nm matrix.
- **vtype** (*str*) – Domain of the variable. Can be any of
  - `'continuous'` – real valued,
  - `'binary'` – either zero or one,
  - `'integer'` – integer valued,
  - `'symmetric'` – symmetric matrix,
  - `'antisym'` – antisymmetric matrix,
  - `'complex'` – complex matrix,
  - `'hermitian'` – complex hermitian matrix,
  - `'semicont'` – zero or real valued and satisfying its bounds (supported by CPLEX and Gurobi only), or
  - `'semiint'` – zero or integer valued and satisfying its bounds (supported by CPLEX and Gurobi only).
- **lower** (anything recognized by *retrieve_matrix*) – A lower bound for the variable.

  Can be either a vector or matrix of the same size as the variable or a scalar that is then broadcasted so that all elements of the variable have the same lower bound.
- **upper** (anything recognized by *retrieve_matrix*) – An upper bound for the variable.

  Can be either a vector or matrix of the same size as the variable or a scalar that is then broadcasted so that all elements of the variable have the same upper bound.

Returns A *Variable* instance.

Example

```
>>> prob=picos.Problem()
>>> x=prob.add_variable('x',3)
>>> x
<3×1 Continuous Variable: x>
```

as_dual()
Returns the Lagrangian dual problem of the problem.

To this end the problem is put in a canonical primal form (see the *note on dual variables*), and the corresponding dual form is returned as a new *Problem*.

as_real()
Returns a modified copy of the problem, where hermitian nn matrices are replaced by symmetric $2n2n$ matrices.

as_socp()

check_current_value_feasibility(*tol=1e-05*, *inttol=0.001*)
Checks whether all variables that appear in constraints are valued and satisfy the constraints up to the given tolerance. In other words, checks whether the variables are valued to form a feasible solution.

Parameters

- **tol** (*float*) – Largest tolerated absolute violation of a constraint. If `None`, the `fesatol` or `tol` solver option is used.

- **inttol** (*float*) – Largest tolerated absolute violation of integrality of an integral variable.

Returns A tuple `(feasible, violation)` where `feasible` is a bool stating whether the solution is feasible and `violation` is either `None`, if `feasible == True`, or the amount of violation, otherwise.

convert_quad_to_socp()
Replaces quadratic constraints with equivalent second order cone constraints.

convert_quadobj_to_constraint()
Replaces a quadratic objective with an equivalent quadratic constraint.

copy()
Creates an independent copy of the problem, using new variables.

Note: Your existing variable, constraint, and metaconstraint references will refer to the original variables, so you cannot query these for solution details after solving the copy. Access the copy's constraints and variables instead.

---

get_constraint(*ind*)
Returns a constraint of the problem, given its index.

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

- If `ind` is an `int` n, then the n-th constraint (starting from 0) is returned, where constraints are counted in the order in which they where passed to the problem.
- if `ind` is a `tuple` (k, i), then the i-th constraint from the k-th group of constraints is returned (both starting from 0). Here *group of constraints* refers to a list of constraints added together via *add_list_of_constraints*.
- If `ind` is a tuple $(k,)$ of length 1, then the k-th group of constraints is returned as a list.

Returns A *constraint* or a list thereof.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(5)]
>>> y=prob.add_variable('y',5)
>>> Cx=prob.add_list_of_constraints([(1|x[i]) < y[i] for i in range(5)])
>>> Cy=prob.add_constraint(y>0)
>>> print(prob) #doctest: +NORMALIZE_WHITESPACE
---------------------
optimization problem (LP):
15 variables, 10 affine constraints
<BLANKLINE>
x   : list of 5 variables, (2, 1), continuous
y   : (5, 1), continuous
<BLANKLINE>
    find vars
such that
  ⟨[1], x[i]⟩ ≤ y[i] ∀ i ∈ [0...4]
  y ≥ 0
---------------------
>>> # Retrieve the 3nd constraint (counted from 0):
>>> prob.get_constraint(1)
<1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>
>>> # Retrieve the 4th consraint from the 1st group:
```

(continues on next page)

(continued from previous page)

```
>>> prob.get_constraint((0,3))
<1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>
>>> # Retrieve the unique constraint of the 2nd 'group':
>>> prob.get_constraint((1,))
<5×1 Affine Constraint: y ≥ 0>
>>> # Retrieve the whole 1st group of constraints:
>>> pprint(prob.get_constraint((0,)))
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <1×1 Affine Constraint: ⟨[1], x[4]⟩ ≤ y[4]>]
```

get_valued_variable(*name*)

Returns the value or values of a variable or of a collection of variables with a common base name.

Parameters **name** (*str*) – Name of a single variable or of a collection of variables (see *get_variable* on how to specify collections).

Raises An exception if any of the variables is not valued, in particular when the problem was not yet solved.

Returns A CVXOPT matrix, if name refers to a single variable, or a list or a dictionary thereof, if the collection of variables specified by name is a list or a dictionary, respectively.

get_variable(*name*)

Returns a single variable with the given name or a list or dictionary of variables with the given name as a common base name. In the latter case the variables must be named name[index] or name[key] with index taken from a set of integer strings and key taken from a set of arbitrary strings.

Parameters **name** (*str*) – Name of a single variable or of a collection of variables.

Returns A *PICOS variable*, if name refers to a single variable, or a list or a dictionary thereof, if the collection of variables specified by name is a list or a dictionary, respectively.

is_complex()

Returns True, if the problem has a complex variable or if there is a complex coefficient or constant inside a constraint.

is_continuous()

Returns True, if all variables are continuous.

is_pure_integer()

Returns True, if all variables are integer.

maximize(*obj*, ***options*)

Sets the objective to maximize the given objective function and calls the solver with the given sequence of options.

Parameters

- **obj** (*Expression*) – The objective function to maximize.
- **options** – A sequence of optional solver options.

Returns A dictionary, see *solve*.

Warning: This is equivalent to *set_objective* followed by *solve* and will thus override any existing objective function and direction.

Further, any supplied options will be stored in the problem as if they were set via *set_option*.

minimize(*obj*, ***options*)
Sets the objective to minimize the given objective function and calls the solver with the given sequence of options.

Parameters

- **obj** (*Expression*) – The objective function to minimize.
- **options** – A sequence of optional solver options.

Returns A dictionary, see *solve*.

Warning: This is equivalent to *set_objective* followed by *solve* and will thus override any existing objective function and direction.

Further, any supplied options will be stored in the problem as if they were set via *set_option*.

obj_value()
Returns the objective value after the problem was solved.

Raises AttributeError, if the problem was not yet solved.

remove_all_constraints()
Removes all constraints from the problem.

This function does not remove bounds set directly on variables; use *remove_all_variable_bounds* to do so.

remove_all_variable_bounds()
Removes all lower and upper bounds from all variables.

remove_constraint(*ind*)
Deletes a constraint from the problem.

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

- If ind is an int n, then the n-th constraint (starting from 0) is deleted, where constraints are counted in the order in which they where passed to the problem.
- if ind is a tuple (k, i), then the i-th constraint from the k-th group of constraints is deleted (both starting from 0). Here *group of constraints* refers to a list of constraints added together via *add_list_of_constraints*.
- If ind is a tuple $(k,)$ of length 1, then the whole k-th group of constraints is deleted.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> from pprint import pprint
>>> prob=pic.Problem()
>>> x=[prob.add_variable('x[{0}]'.format(i),2) for i in range(4)]
>>> y=prob.add_variable('y',4)
>>> Cxy=prob.add_list_of_constraints([(1|x[i])<y[i] for i in range(4)])
>>> Cy=prob.add_constraint(y>0)
>>> Cx0to2=prob.add_list_of_constraints([x[i]<2 for i in range(3)])
>>> Cx3=prob.add_constraint(x[3]<1)
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[1]⟩ ≤ y[1]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
```

(continues on next page)

(continued from previous page)

```
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <4×1 Affine Constraint: y ≥ 0>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 2nd constraint (counted from 0):
>>> prob.remove_constraint(1)
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <4×1 Affine Constraint: y ≥ 0>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 2nd group of constraints, i.e. the constraint y > 0:
>>> prob.remove_constraint((1,))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>,
 <2×1 Affine Constraint: x[3] ≤ [1]>]
>>> # Delete the 3rd remaining group of constraints, i.e. x[3] < [1]:
>>> prob.remove_constraint((2,))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[1] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>]
>>> # Delete 2nd constraint of the 2nd remaining group, i.e. x[1] < |2|:
>>> prob.remove_constraint((1,1))
>>> pprint(prob.constraints) #doctest: +NORMALIZE_WHITESPACE
[<1×1 Affine Constraint: ⟨[1], x[0]⟩ ≤ y[0]>,
 <1×1 Affine Constraint: ⟨[1], x[2]⟩ ≤ y[2]>,
 <1×1 Affine Constraint: ⟨[1], x[3]⟩ ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>]
```

remove_variable(*name*)

Removes a variable from the problem.

Parameters **name** (*str*) – Name of the variable to remove.

> **Warning:** This method does not check if some constraint still involves the variable to be removed.

reset(*resetOptions=False*)

Resets the problem instance to its initial empty state.

Parameters **resetOptions** (*bool*) – Whether also solver options should be reset to their default values.

reset_solver_instances()

Resets all solver instances, so that the problem will be reimported and solved from scratch.

set_all_options_to_default()
Sets all solver options to their default value.

set_objective(*typ*, *expr*)
Sets the objective function and optimization direction of the problem.

Parameters

- **typ** (*str*) – Can be either `'max'`, `'min'`, or `'find'`, for a maximization, minimization, and feasibility problem, respectively.
- **expr** (*Expression*) – The objective function to be minimized or maximized. This parameter is ignored if `typ == 'find'`.

set_option(*key*, *val*)
Sets a single solver option to the given value.

Parameters

- **key** (*str*) – String name of the option, see below for a list.
- **val** – New value for the option.

The following options are available and are listed with their default values.

- General options common to all solvers:
  - `strict_options = False` – If `True`, unsupported general options will raise an *UnsupportedOptionError* exception, instead of printing a warning.
  - `verbose = 1` – Verbosity level.
    - `-1` attempts to suppress all output, even errors.
    - `0` only outputs warnings and errors.
    - `1` generates standard informative output.
    - `2` prints all available information for debugging purposes.
  - `allow_license_warnings = True` – Whether solvers are allowed to ignore the `verbose` option to print licensing related warnings.

    Using this option to surpress licensing related warnings is done at your own legal responsibility.
  - `solver = None` – Solver to use.
    - `None` lets PICOS select a suitable solver for you.
    - `'cplex'` for CPLEX.
    - `'cvxopt'` for CVXOPT.
    - `'glpk'` for GLPK.
    - `'mosek'` for MOSEK.
    - `'gurobi'` for Gurobi.
    - `'scip'` for SCIP (formerly ZIBOpt).
    - `'smcp'` for SMCP.
  - `tol = 1e-8` – Relative gap termination tolerance for interior-point optimizers (feasibility and complementary slackness).

    This option is currently ignored by GLPK. SCIP will only lower its precision for large values and not increase it for small ones.
  - `maxit = None` – Maximum number of iterations for simplex or interior-point optimizers). *Currently ignored by SCIP.*

- `lp_root_method = None` – Algorithm used to solve continuous linear problems, including the root relaxation of mixed integer problems.

  * `None` lets PICOS or the solver select it fo you.
  * `'psimplex'` for primal Simplex.
  * `'dsimplex'` for dual Simplex.
  * `'interior'` for the interior point method.

  This option currently works only with CPLEX, Gurobi and MOSEK. With GLPK it works for LPs but not for the MIP root relaxation.

- `lp_node_method = None` – Algorithm used to solve subproblems at non-root nodes of the branching tree built when solving mixed integer programs.

  * `None` lets PICOS or the solver select it fo you.
  * `'psimplex'` for primal Simplex.
  * `'dsimplex'` for dual Simplex.
  * `'interior'` for the interior point method.

  This option currently works only with CPLEX, Gurobi and MOSEK.

- `timelimit = None` – Total time limit for the solver, in seconds. The default `None` means no time limit.

  This option is not supported by CVXOPT and SMCP.

- `treememory = None` – Bound on the memory used by the branch and bound tree, in Megabytes.

  This option currently works only with CPLEX and SCIP.

- `gaplim = 1e-4` – For mixed integer problems, the solver returns a solution as soon as this value for the relative gap between the primal and the dual bound is reached.

- `noprimals = False` – If `True`, do not retrieve a primal solution from the solver.

- `noduals = False` – If `True`, do not retrieve optimal values for the dual variables. This can speed up solvers that do not produce a dual solution as part of their primal solution process.

- `nbsol = None` – Maximum number of feasible solution nodes visited when solving a mixed integer problem, before returning the best one found.

  If you want to obtain all feasible solutions that the solver encountered, use `pool_size` instead.

- `pool_size = None` – Maximum number of mixed integer feasible solutions returned, instead of just a single one.

  If you merely want to set a limit on the number of feastble solution nodes that are visited, use `nbsol` instead.

  This option currently works only with CPLEX.

- `pool_absgap = None` – Discards solutions from the solution pool as soon as a better solution is found that beats it by the given absolute gap tolerance with respect to the objective function.

  This option currently works only with CPLEX.

- `pool_relgap = None` – Discards solutions from the solution pool as soon as a better solution is found that beats it by the given relative gap tolerance with respect to the objective function.

  This option currently works only with CPLEX.

- `hotstart = False` – If *True*, tells the mixed integer optimizer to start from the (partial) solution specified in the variables' *value* attributes.

  This option currently works only with CPLEX, Gurobi and MOSEK.

- `solve_via_dual = None` – If set to `True`, the Lagrangian Dual (computed with the function *as_dual*) is passed to the solver, instead of the problem itself. In some scenarios this can yield a signficant speed-up. If set to `None`, PICOS chooses automatically whether the problem itself or its dual should be passed to the solver.

- 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`.

  - `uboundlimit = None` – Tells CPLEX to stop as soon as an upper bound smaller than this value is found.
  - `lboundlimit = None` – Tells CPLEX to stop as soon as a lower bound larger than this value is found.
  - `boundMonitor = True` – Tells CPLEX to store information about the evolution of the bounds during the solving process. At the end of the computation, a list of triples *(time,lowerbound,upperbound)* will be provided in the field bounds_monitor‘ of the dictionary returned by *solve*.
- Specific options available for CVXOPT, SMCP and ECOS:
  - `feastol = None` – Feasibility tolerance passed to cvx.solvers.options If `None`, then the value of the option `tol` is used.
  - `abstol = None` – Absolute tolerance passed to cvx.solvers.options If `None`, then the value of the option `tol` is used.
  - `reltol = None` – relative tolerance passed to cvx.solvers.options If `None`, then **ten times** the value of the option `tol` is used.
- 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.
- Specific options available for MOSEK:
  - `mosek_params = {}` – A dictionary of MOSEK Fusion API parameters to be set before the MOSEK optimizer is called.
- Specific options available for SCIP:
  - `scip_params = {}` – A dictionary of SCIP parameters to be set before the SCIP optimizer is called.

    For example, `scip_params = {'lp/threads': 4}` sets the number of threads to solve LPs with to `4`.

Note: Options can also be passed as a parameter sequence of the form `key = value` when the *Problem* is created or later to the function *solve*.

set_var_value(*name*, *value*, *optimalvar=False*)

Sets the *value* of the given variable.

Parameters

- **or tuple name** (*str*) – Name of the variable.
- **value** (Anything recognized by *retrieve_matrix*) – The value to be set.

Example

```
>>> prob=picos.Problem()
>>> x=prob.add_variable('x', 2)
>>> prob.set_var_value('x', [3,4]) # equivalent to x.value = [3,4]
>>> abs(x)**2
<Quadratic Expression: ‖x‖^2>
>>> print(abs(x)**2)
25.0
```

Note: The `hotstart` option allows certain solvers to leverage variables that were valued manually or by a preceding solution search.

solve(***options*)

Hands the problem to a solver.

You can select the solver manually with the `solver` option. Otherwise a suitable solver will be selected among those that are available on the platform.

Once the problem has been solved, the optimal solution can be obtained by querying the *value* property of the variables and the optimal dual values can be accessed via the *dual* property of the constraints.

Parameters **options** – A sequence of optional solver options. In particular, you can use this to select a solver via the `solver` option.

Returns

A dictionary that contains the following common entries, and potentially further solver-specific or option-specific fields:

- `'status'` – The solution status as a human readable string, such as `'optimal'` or `'infeasible'`. The exact wording and available phrases depend on the solver being used.
- `'time'` – The time spent searching for a solution in seconds, *excluding* any overhead produced by PICOS when exporting the problem or configuring the solver.
- `'primals'` – A dictionary mapping PICOS variables to their value in the solution produced by the solver.
- `'duals'` – A list of dual values produced by the solver, in the order in which the constraints were added.

Warning: Any supplied options will be stored in the problem as if they were set via *set_option*.

Note: If the problem is dualized or cast as a SOCP during solution search, then it will be solved from scratch upon subsequent searches, even if the solver supports problem updates efficiently.

update_options(***options*)

Sets multiple solver options at once.

Parameters **options** – A parameter sequence of the form `key = value`.

For a list of available options and their default values, see the documentation of *set_option*.

verbosity()

Returns The problem's current verbosity level.

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

Parameters

- **filename** (*str*) – Path and name of the output file. The export format is inferred from the file extension. Supported extensions and their associated format are:
  - `'.cbf'` – Conic Benchmark Format.

    This format is suitable for optimization problems involving second order and/or semidefinite cone constraints. This is a standard choice for conic optimization problems. Visit the website of The Conic Benchmark Library or read A benchmark library for conic mixed-integer and continuous optimization by Henrik A. Friberg for more information.
  - `'.lp'` – LP format.

    This format handles only linear constraints, unless the writer `'cplex'` is used. In the latter case the extended CPLEX LP format is used instead.
  - `'.mps'` – MPS format.

    As the writer, you need to choose one of `'cplex'`, `'gurobi'` or `'mosek'`.
  - `'.opf'` – OPF format.

    As the writer, you need to choose `'mosek'`.
  - `'.dat-s'` – Sparse SDPA format.

    This format is suitable for semidefinite programs. Second order cone constraints are stored as semidefinite constraints on an *arrow shaped* matrix.
- **writer** (*str*) – The default `'picos'` denotes PICOS' internal writer, which can export to *LP*, *CBF*, and *Sparse SDPA* formats. If CPLEX, Gurobi or MOSEK is installed, you can choose `'cplex'`, `'gurobi'`, or `'mosek'`, respectively, to make use of that solver's export function and get access to more formats.

> **Warning:** For problems involving a symmetric matrix variable X (typically, semidefinite programs), the expressions involving X are stored in PICOS as a function of $svec(X)$, the symmetric vectorized form of X (see Dattorro, ch.2.2.2.1), and are also exported in that form. As a result, using an external solver on a problem description file exported by PICOS will also yield a solution in this symmetric vectorized form.
>
> The CBF writer tries to write symmetric variables X in the section `PSDVAR` of the .cbf file. However, this is possible only if the constraint $X \succeq 0$ appears in the problem, and no other LMI involves X. If these two conditions are not satisfied, then the symmetric vectorization of X is used as a (free) variable of the section `VAR` in the .cbf file, as explained in the previous paragraph.

options

status
The solution status of the problem.

type
The problem type as a string, such as 'LP', 'MILP' or 'SOCP'.

2.4.6 picos.solvers

This package contains the interfaces to the optimization solvers that PICOS uses as its backend. You do not need to instanciate any of the solver classes directly; if you want to select a particular solver, it is most convenient to supply it to *Problem.solve* via the `solver` keyword argument.

Functions

all_solvers()	**returns** A dictionary mapping solver names to implementation classes.
available_solvers([problem])	**returns** A list of names of available solvers.
get_solver(name)	**returns** Implementation class of the solver of the given name.
potential_solvers(problem)	**returns** A list of names of solvers that support the given problem.
suggested_solver(problem[, order, returnClass])	**returns** The name or class of an available solver that can handle the given
supportLevelString(level)	

all_solvers

picos.solvers.**all_solvers**()

Returns A dictionary mapping solver names to implementation classes.

available_solvers

picos.solvers.**available_solvers**(*problem=None*)

Returns A list of names of available solvers.

Parameters **problem** (picos.Problem) – Return only solvers that also support this problem.

get_solver

picos.solvers.**get_solver**(*name*)

Returns Implementation class of the solver of the given name.

potential_solvers

picos.solvers.**potential_solvers**(*problem*)

Returns A list of names of solvers that support the given problem.

suggested_solver

picos.solvers.**suggested_solver**(*problem, order=['gurobi', 'cplex', 'mosek', 'mskfsn', 'scip', 'ecos', 'glpk', 'smcp', 'cvxopt'], returnClass=False*)

Returns The name or class of an available solver that can handle the given problem type.

Parameters

- **order** (*list*) – The order in which solvers are considered, as a list of solver names. If the list does not contain every solver it will be extended arbitrarily to do so.
- **returnClass** (*bool*) – Whether to return the solver's class instead of its keyword name.

supportLevelString

picos.solvers.**supportLevelString**(*level*)

Classes

CPLEXSolver(problem)	Implementation of the CPLEX solver.
CVXOPTSolver(problem)	
ConflictingOptionsError	An exception raised by implementations of *_solve* to signal to the user that two options they specified cannot be used in conjunction.
DependentOptionError	An exception raised by implementations of *_solve* to signal to the user that an option they specified needs another option to also be set.
ECOSSolver(problem)	
GLPKSolver(problem)	
GurobiSolver(problem)	
InappropriateSolverError	An exception raised by implementations of *_solve* to signal to the user that the solver (or its requested subsolver) does not support the given problem type.
MOSEKFusionSolver(problem)	
MOSEKSolver(problem)	
NoAppropriateSolverError	An exception raised when no fitting solver is available.
OptionError	Base class for solver option related errors.
OptionValueError	An exception raised by implementations of *_solve* to signal to the user that they have set an option to an invalid value.
ProblemUpdateError	An exception raised by implementations of *_update_problem* to signal to the method *_load_problem* that the problem needs to be re-imported.
SCIPSolver(problem)	
SMCPSolver(problem)	
Solver(problem, displayName, longDisplayName)	Abstract base class for a wrapper around the internal problem representation of solvers.
SolverError	Base class for solver-specific exceptions.
UnsupportedOptionError	An exception raised by implementations of *_solve* to signal to the user that an option they specified is not supported by the solver or the requested sub-solver, or in conjunction with the given problem type or with another option.

CPLEXSolver

class picos.solvers.**CPLEXSolver**(*problem*)

Bases: *picos.solvers.Solver*

Implementation of the CPLEX solver.

Note: Names are used instead of indices for identifying both variables and constraints since indices can change if the CPLEX instance is modified.

Attributes Summary

DEFAULT_HEADER_WIDTH	

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.

Support levels are used for determining a solver's priority when PICOS selects a solver, and for skipping tests that are known/likely to fail.

Returns A number indicating how well the problem is supported, which must be one of the *SUPPORT_LEVEL_** constants.

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

CVXOPTSolver

class picos.solvers.**CVXOPTSolver**(*problem*)

Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()
Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

ConflictingOptionsError

exception picos.solvers.**ConflictingOptionsError**
Bases: *picos.solvers.OptionError*

An exception raised by implementations of *_solve* to signal to the user that two options they specified cannot be used in conjunction.

DependentOptionError

exception picos.solvers.**DependentOptionError**
Bases: *picos.solvers.OptionError*

An exception raised by implementations of *_solve* to signal to the user that an option they specified needs another option to also be set.

ECOSSolver

class picos.solvers.**ECOSSolver**(*problem*)
Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH	
array	
ecos	Returns the ECOS core module (found in ecos.py), which is obtained by `import ecos` up to ECOS 2.0.6 and by `import ecos.ecos` starting with ECOS 2.0.7.
matrix	

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
stack(*args)	Stacks vectors or matrices, the latter vertically.
support_level(problem)	Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.
zeros(shape)	Creates a zero array or a zero matrix, depending on `shape`.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

array

ecos

Returns the ECOS core module (found in ecos.py), which is obtained by `import ecos` up to ECOS 2.0.6 and by `import ecos.ecos` starting with ECOS 2.0.7.

matrix

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()
Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

stack(**args*)
Stacks vectors or matrices, the latter vertically.

classmethod support_level(*problem*)
Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.

Support levels are used for determining a solver's priority when PICOS selects a solver, and for skipping tests that are known/likely to fail.

Returns A number indicating how well the problem is supported, which must be one of the *SUPPORT_LEVEL_** constants.

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

zeros(*shape*)
Creates a zero array or a zero matrix, depending on `shape`.

GLPKSolver

class picos.solvers.**GLPKSolver**(*problem*)
Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH	

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.

Support levels are used for determining a solver's priority when PICOS selects a solver, and for skipping tests that are known/likely to fail.

Returns A number indicating how well the problem is supported, which must be one of the *SUPPORT_LEVEL_** constants.

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

GurobiSolver

class picos.solvers.**GurobiSolver**(*problem*)

Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()
Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)
Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.

Support levels are used for determining a solver's priority when PICOS selects a solver, and for skipping tests that are known/likely to fail.

Returns A number indicating how well the problem is supported, which must be one of the *SUPPORT_LEVEL_** constants.

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

InappropriateSolverError

exception picos.solvers.**InappropriateSolverError**
Bases: *picos.solvers.SolverError*

An exception raised by implementations of *_solve* to signal to the user that the solver (or its requested sub-solver) does not support the given problem type.

MOSEKFusionSolver

class picos.solvers.**MOSEKFusionSolver**(*problem*)
Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()
Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

MOSEKSolver

class picos.solvers.**MOSEKSolver**(*problem*)
Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH	
env	This references a MOSEK environment, which is shared among all MOSEKSolver instances.

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the

Continued on next page

Table 64 – continued from previous page

reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

env

This references a MOSEK environment, which is shared among all MOSEKSolver instances. (The MOSEK documentation states that "[a]ll tasks in the program should share the same environment.")

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

NoAppropriateSolverError

exception picos.solvers.**NoAppropriateSolverError**
Bases: Exception

An exception raised when no fitting solver is available.

OptionError

exception picos.solvers.**OptionError**
Bases: *picos.solvers.SolverError*

Base class for solver option related errors.

OptionValueError

exception picos.solvers.**OptionValueError**
Bases: *picos.solvers.OptionError*, ValueError

An exception raised by implementations of *_solve* to signal to the user that they have set an option to an invalid value.

ProblemUpdateError

exception picos.solvers.**ProblemUpdateError**
Bases: *picos.solvers.SolverError*

An exception raised by implementations of *_update_problem* to signal to the method *_load_problem* that the problem needs to be re-imported.

SCIPSolver

class picos.solvers.**SCIPSolver**(*problem*)
Bases: *picos.solvers.Solver*

Attributes Summary

DEFAULT_HEADER_WIDTH	

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but

Continued on next page

Table 66 – continued from previous page

problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

SMCPSolver

class picos.solvers.**SMCPSolver**(*problem*)

Bases: *picos.solvers.CVXOPTSolver*

Attributes Summary

DEFAULT_HEADER_WIDTH

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	
solve()	Solves the problem and returns the solution.
support_level(problem)	
supported_constraints()	
supported_objectives()	
supports_integer()	
supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

classmethod supported_constraints()

classmethod supported_objectives()

classmethod supports_integer()

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()

verbosity()

Returns The problem's current verbosity level.

Solver

class picos.solvers.**Solver**(*problem*, *displayName*, *longDisplayName*)

Bases: abc.ABC

Abstract base class for a wrapper around the internal problem representation of solvers.

Creates an instance of a wrapper around a solver's internal problem representation of the given PICOS problem formulation.

An exception is raised when the solver is not available on the user's platform. No exception is raised when the problem type is not supported as the problem is first imported when the user calls *solve*.

Solver implementations are supposed to also implement *__init__*, but with *problem* as its only positional argument, and using *super* to provide fixed values for this method's additional parameters.

Parameters

- **problem** (*Problem*) – A PICOS optimization problem.
- **displayName** (*str*) – The short display name of the solver.
- **longDisplayName** (*str*) – The long display name of the solver.

Attributes Summary

DEFAULT_HEADER_WIDTH	

Methods Summary

available([verbose])	**returns** Whether the solver is properly installed on the system.
external_problem()	**returns** The external (PICOS) problem represenation.
internal_problem()	**returns** The solver's internal problem represenation.
needs_quad_to_socp_cast()	**returns** Whether second order cone constraints are supported but
problem_support_level()	**returns** How well the problem in its current state is supperted by the
reset_problem()	Resets the solver's internal problem representation and related data.
solve()	Solves the problem and returns the solution.
support_level(problem)	Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.
supported_constraints()	**returns** All constraint classes that the solver can import.
supported_objectives()	**returns** All objective function types that the solver can import.
supports_integer()	**returns** Whether (mixed) integer problems are supported.

Continued on next page

Table 70 – continued from previous page

supports_quad_socp_mix()	**returns** Whether quadratic constraints and (rotated) second order cone
test_availability()	Checks whether the solver is properly installed on the system, and raises an appropriate exception (usually *ModuleNotFoundError* or *ImportError*) if not.
verbosity()	**returns** The problem's current verbosity level.

Attributes Documentation

DEFAULT_HEADER_WIDTH = 35

Methods Documentation

classmethod available(*verbose=False*)

Returns Whether the solver is properly installed on the system.

external_problem()

Returns The external (PICOS) problem represenation.

internal_problem()

Returns The solver's internal problem represenation.

classmethod needs_quad_to_socp_cast()

Returns Whether second order cone constraints are supported but quadratic problems are not.

problem_support_level()

Returns How well the problem in its current state is supperted by the solver, as a nonnegative integer.

reset_problem()

Resets the solver's internal problem representation and related data.

Method implementations are supposed to

- set *int* to None (after performing any garbage collection), and
- reset all additional problem metadata to the state it had after *__init__*, in particular the data stored for *_update_problem*.

Solver implementations should not call *reset_problem* directly, except from within *__init__* if this is convenient.

The user may call this method at any time if they wish to solve the problem from scratch.

solve()

Solves the problem and returns the solution.

Returns A quadruple (primals, duals, objectiveValue, meta).

classmethod support_level(*problem*)

Solver implementations may overwrite this method if necessary, for instance to indicate experimental or limited support, or to disallow certain combinations of constraints that are supported individually, or to allow constraints that are otherwise not supported if they originate from a metaconstraint that is supported directly.

Support levels are used for determining a solver's priority when PICOS selects a solver, and for skipping tests that are known/likely to fail.

Returns A number indicating how well the problem is supported, which must be one of the *SUPPORT_LEVEL_** constants.

classmethod supported_constraints()

Returns All constraint classes that the solver can import.

classmethod supported_objectives()

Returns All objective function types that the solver can import.

classmethod supports_integer()

Returns Whether (mixed) integer problems are supported.

classmethod supports_quad_socp_mix()

Returns Whether quadratic constraints and (rotated) second order cone constraints may appear in the same problem.

classmethod test_availability()
Checks whether the solver is properly installed on the system, and raises an appropriate exception (usually *ModuleNotFoundError* or *ImportError*) if not. Does not return anything.

verbosity()

Returns The problem's current verbosity level.

SolverError

exception picos.solvers.**SolverError**
Bases: Exception

Base class for solver-specific exceptions.

UnsupportedOptionError

exception picos.solvers.**UnsupportedOptionError**
Bases: *picos.solvers.OptionError*

An exception raised by implementations of *_solve* to signal to the user that an option they specified is not supported by the solver or the requested sub-solver, or in conjunction with the given problem type or with another option. If the option is valid but not supported by PICOS, then NotImplementedError should be raised instead. The exception is only raised if the *strictOptions* option is set, otherwise a warning is printed.

Variables

SUPPORT_LEVEL_EXPERIMENTAL	int([x]) -> integer int(x, base=10) -> integer
SUPPORT_LEVEL_LIMITED	int([x]) -> integer int(x, base=10) -> integer
SUPPORT_LEVEL_NATIVE	int([x]) -> integer int(x, base=10) -> integer
SUPPORT_LEVEL_NONE	int([x]) -> integer int(x, base=10) -> integer
SUPPORT_LEVEL_SECONDARY	int([x]) -> integer int(x, base=10) -> integer
order	The default preference list for solver selection.

SUPPORT_LEVEL_EXPERIMENTAL

picos.solvers.**SUPPORT_LEVEL_EXPERIMENTAL = 2**
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

SUPPORT_LEVEL_LIMITED

picos.solvers.**SUPPORT_LEVEL_LIMITED = 1**
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

SUPPORT_LEVEL_NATIVE

picos.solvers.**SUPPORT_LEVEL_NATIVE = 4**
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

SUPPORT_LEVEL_NONE

picos.solvers.**SUPPORT_LEVEL_NONE = 0**
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

SUPPORT_LEVEL_SECONDARY

picos.solvers.**SUPPORT_LEVEL_SECONDARY = 3**
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

order

picos.solvers.**order = ['gurobi', 'cplex', 'mosek', 'mskfsn', 'scip', 'ecos', 'glpk', 'sm**
The default preference list for solver selection. Solvers that do not appear are appended arbitrarily when selecting a solver.

The order is chosen as follows:

- Commercial solvers appear first as the user has spent money or academic licensing effort to make them available and is likely to want them used.
- MOSEK's high level Fusion API was found to be a performance bottleneck (2018-10), so it appears at the end of the commercial solver list (so that MOSEK's low level Optimizer API takes precedence).
- Commercial solvers are sorted based on LP benchmark results in http://plato.asu.edu/talks/informs2017.pdf as LPs are the most basic problem type supported by PICOS and the benchmark results appear decisive.
- CVXOPT appears last as it is the only solver that PICOS depends on and thus presence on the system is least likely to express user preference.
- The remaining noncommercial solvers are sorted based on the PICOS maintainers' subjectively perceived impression of "maintainedness".

2.4.7 picos.tools

Many of the tools, in particular the algebraic functions, are also available in the *picos* namespace. For example, you can write *picos.sum* instead of *picos.tools.sum*. In the future, we are looking to split the toolbox into mutltiple modules, so that it is clear which of the functions are imported into the *picos* namespace.

Members

exception picos.tools.**DualizationError**(*msg*)
Bases: Exception

Exception raised when a non-standard conic problem is being dualized.

exception picos.tools.**NonConvexError**(*msg*)
Bases: Exception

Exception raised when non-convex quadratic constraints are passed to a solver which cannot handle them.

exception picos.tools.**NotAppropriateSolverError**(*msg*)
Bases: Exception

Exception raised when trying to solve a problem with a solver which cannot handle it

exception picos.tools.**QuadAsSocpError**(*msg*)
Bases: Exception

Exception raised when the problem can not be solved in the current form, because quad constraints are not handled. User should try to convert the quads as socp.

class picos.tools.**NonWritableDict**
Bases: dict

picos.tools.**ball**(*r*, *p=2*)
returns a *Ball* object representing:

- a L_p Ball of radius r ($\{x : \|x\|_p \geq r\}$) if $p \geq 1$
- the convex set $\{x \geq 0 : \|x\|_p \geq r\}$ $p < 1$.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.ball(2,3)  #doctest: +NORMALIZE_WHITESPACE
<p-Norm Constraint: ‖x‖_3 ≤ 2>
>>> x << pic.ball(1,0.5)
<Generalized p-Norm Constraint: ‖x‖_(1/2) ≥ 1>
```

picos.tools.**blocdiag**(*X*, *n*)
makes diagonal blocs of X, for indices in [sub1,sub2[n indicates the total number of blocks (horizontally)

picos.tools.**block_idx**(*i*, *sizes*)

picos.tools.**break_cols**(*mat*, *sizes*)

picos.tools.**break_rows**(*mat*, *sizes*)

picos.tools.**copy_exp_to_new_vars**(*exp*, *cvars*, *complex=None*)

picos.tools.**cplx_vecmat_to_real_vecmat**(*M*, *sym=True*, *times_i=False*)

If the columns of M are vectorizations of matrices of the form A +iB:

- If times_i is False (default), return vectorizations of the block matrix [A, -B; B, A] otherwise, return vectorizations of the block matrix [-B, -A; A, -B].
- If sym=True, returns the columns with respect to the sym-vectorization of the variables of the LMI.

picos.tools.**detect_range**(*sequence*, *asQuadruple=False*, *asStringTemplate=False*, *shortString=False*)
Detects a Python range object yielding the same sequence as the given integer sequence.

By default, returns a range object mirroring the input sequence.

Parameters

- **sequence** – An integer sequence that can be mirrored by a Python range.
- **asQuadruple** (*bool*) – Whether to return a quadruple with factor, inner shift, outer shift, and length, formally `(a, i, o, n)` such that `[a*(x+i)+o for x in range(n)]` mirrors the input sequence.
- **asStringTemplate** (*bool*) – Whether to return a format string that, if instanciated with numbers from `0` to `len(sequence) - 1`, yields math expression strings that describe the input sequence members.
- **shortString** (*bool*) – Whether to return condensed string templates that are designed to be instanciated with an index character string. Requires asStringTemplate to be `True`.

Raises

- **TypeError** – If the input is not an integer sequence.
- **ValueError** – If the input cannot be mirrored by a Python range.

Returns A range object, a quadruple of numbers, or a format string.

Example

```
>>> from picos.tools import detect_range as dr
>>> R = range(7,30,5)
>>> S = list(R)
>>> S
[7, 12, 17, 22, 27]
>>> # By default, returns a matching range object:
>>> dr(S)
range(7, 28, 5)
>>> dr(S) == R
True
>>> # Sequence elements can also be decomposed w.r.t. range(len(S)):
>>> a, i, o, n = dr(S, asQuadruple=True)
>>> [a*(x+i)+o for x in range(n)] == S
True
>>> # The same decomposition can be returned in a string representation:
>>> dr(S, asStringTemplate=True)
```

(continues on next page)

(continued from previous page)

```
'5·({} + 1) + 2'
>>> # Short string representations are designed to accept index names:
>>> dr(S, asStringTemplate=True, shortString=True).format("i")
'5(i+1)+2'
>>> dr(range(0,100,5), asStringTemplate=True, shortString=True).format("i")
'5i'
>>> dr(range(10,100), asStringTemplate=True, shortString=True).format("i")
'i+10'
```

Example

```
>>> # This works with decreasing ranges as well.
>>> R2 = range(10,4,-2)
>>> S2 = list(R2)
>>> S2
[10, 8, 6]
>>> dr(S2)
range(10, 5, -2)
>>> dr(S2) == R2
True
>>> a, i, o, n = dr(S2, asQuadruple=True)
>>> [a*(x+i)+o for x in range(n)] == S2
True
>>> T = dr(S2, asStringTemplate=True, shortString=True)
>>> [T.format(i) for i in range(len(S2))]
['-2(0-5)', '-2(1-5)', '-2(2-5)']
```

picos.tools.**detrootn**(*exp*)

returns a *DetRootN_Exp* object representing the determinant of the n th-root of the symmetric matrix `exp`, where n is the dimension of the matrix. This can be used to enter constraints of the form $(\det X)^{1/n} \geq t$. Note that X is forced to be positive semidefinite when a constraint of this form is entered in PICOS. Determinant inequalities are internally reformulated as a set of Linear Matrix Inequalities (SDP).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> t < pic.detrootn(X)
<n-th Root of a Determinant Constraint: det(X)^(1/3) ≥ t>
```

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.diag(x-y,4)
<4×4 Affine Expression: Diag(x - y)>
```

(continues on next page)

(continued from previous page)

```
>>> pic.diag(x//y)
<2×2 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.diag_vect(X)
<3×1 Affine Expression: diag(X)>
```

picos.tools.**drawGraph**(*G*, *capacity='capacity'*)
"Draw a given Graph

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.**exp**(*x*)
Exponentiation of a PICOS, CVXOPT, NumPy, or numeric expression.

picos.tools.**expcone**()
returns a *ExponentialCone* object representing the set closure $\{(x, y, z) : y > 0, y \exp(x/y) \leq z\}$

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> pic.expcone()
<Exponential Cone: cl{[x; y; z] : y·exp(z/y) ≤ x, x > 0, y > 0}>
>>> x << pic.expcone()
<Exponential Cone Constraint: x[0] ≥ x[1]·exp(x[2]/x[1])>
```

picos.tools.**flatten**(*l*)
flatten a (recursive) list of list

picos.tools.**flow_Constraint**(*G*, *f*, *source*, *sink*, *flow_value*, *capacity=None*, *graphName=''*)
Constructs a network flow constraint.

Parameters

- **G** (networkx DiGraph.) – A directed graph.
- **f** (*dict*) – A dictionary of variables indexed by the edges of `G`.
- **source** – Either a node of `G` or a list of nodes in case of a multi-source flow.
- **sink** – Either a node of `G` or a list of nodes in case of a multi-sink flow.
- **flow_value** – The value of the flow, or a list of values in case of a single-source/multi-sink flow. In the latter case, the values represent the demands of each sink (resp. of each source for a multi-source/single-sink flow). The values can be either constants or *affine expressions*.
- **capacity** – Either `None` or a string. If this is a string, it indicates the key of the edge dictionaries of `G` that is used for the capacity of the links. Otherwise, edges have an unbounded capacity.
- **graphName** (*str*) – Name of the graph as used in the string representation of the constraint.

picos.tools.**geomean**(*exp*)

returns a *GeoMeanExp* object representing the geometric mean of the entries of `exp[:]`. This can be used to enter inequalities of the form `t <= geomean(x)`. Note that geometric mean inequalities are internally reformulated as a set of SOC inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',1)
>>> y = prob.add_variable('y',3)
>>> # Add the constraint x <= (y0*y1*y2)**(1./3) to the problem:
>>> prob.add_constraint(x<pic.geomean(y))
<Geometric Mean Constraint: x ≤ geomean(y)>
```

picos.tools.**import_cbf**(*filename*)

Imports the data from a CBF file, and creates a *Problem* object.

The created problem contains one (multidimmensional) variable for each cone specified in the section `VAR` of the .cbf file, and one (multidimmensional) constraint for each cone specified in the sections `CON` and `PSDCON`.

Semidefinite variables defined in the section `PSDVAR` of the .cbf file are represented by a matrix picos variable `X` with `X.vtype = 'symmetric'`.

This function returns a tuple `(P,x,X,data)`, where:

- `P` is the imported picos *Problem* object.
- `x` is a list of *Variable* objects, representing the (multidimmensional) scalar variables.
- `X` is a list of *Variable* objects, representing the symmetric semidefinite positive variables.
- `data` is a dictionary containing picos parameters (*AffinExp* objects) used to define the problem. Indexing is with respect to the blocks of variables as defined in the sections `VAR` and `CON` of the .cbf file.

picos.tools.**is_idty**(*mat*, *vtype='continuous'*)

picos.tools.**is_integer**(*x*)

picos.tools.**is_numeric**(*x*)

picos.tools.**is_realvalued**(*x*)

picos.tools.**kron**(*A*, *B*)

Kronecker product of 2 expression, at least one of which must be constant

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> import numpy as np
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,3))
>>> X.value = cvx.matrix(range(12),(4,3))
>>> I = pic.new_param('I',np.eye(2))
>>> print(pic.kron(I,X)) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  0.00e+00  0.00e+00  0.00e+00]
[ 1.00e+00  5.00e+00  9.00e+00  0.00e+00  0.00e+00  0.00e+00]
[ 2.00e+00  6.00e+00  1.00e+01  0.00e+00  0.00e+00  0.00e+00]
[ 3.00e+00  7.00e+00  1.10e+01  0.00e+00  0.00e+00  0.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  4.00e+00  8.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  1.00e+00  5.00e+00  9.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  2.00e+00  6.00e+00  1.00e+01]
[ 0.00e+00  0.00e+00  0.00e+00  3.00e+00  7.00e+00  1.10e+01]
```

picos.tools.**kullback_leibler**(*x*, *y=None*)
A shorthand for *KullbackLeibler*.

If the second optional argument is passed, the resulting expression is the Kullback-Leibler divergence $\sum_i x_i \log(x_i/y_i)$, otherwise it is the (negative) entropy $\sum_i x_i \log(x_i)$.

picos.tools.**lambda_max**(*exp*)
largest eigenvalue of a square matrix expression (cf. *pic.sum_k_largest(exp,1)*)

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> pic.lambda_max(X) < 2
<Sum of Largest Eigenvalues Constraint: λ_max(X) ≤ 2>
```

picos.tools.**lambda_min**(*exp*)
smallest eigenvalue of a square matrix expression (cf. *pic.sum_k_smallest(exp,1)*)

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> pic.lambda_min(X) > -1
<Sum of Smallest Eigenvalues Constraint: λ_min(X) ≥ -1>
```

picos.tools.**log**(*x*)
The logarithm of a PICOS, CVXOPT, NumPy, or numeric expression.

picos.tools.**logsumexp**(*exp*)
A shorthand for *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
<LSE Constraint: logosumoexp(A·x) ≤ 0>
```

picos.tools.**lowtri**(*exp*)
if `exp` is a square affine expression of size (n,n), `lowtri(exp)` returns the (n(n+1)/2)-vector of the lower triangular elements of `exp`.

Example

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> prob=pic.Problem()
>>> X=prob.add_variable('X',(4,4),'symmetric')
>>> pic.tools.lowtri(X)
<10×1 Affine Expression: lowtri(X)>
>>> X0 = cvx.matrix(range(16),(4,4))
>>> X.value = X0 * X0.T
>>> print(X) #doctest: +NORMALIZE_WHITESPACE
[ 2.24e+02  2.48e+02  2.72e+02  2.96e+02]
[ 2.48e+02  2.76e+02  3.04e+02  3.32e+02]
[ 2.72e+02  3.04e+02  3.36e+02  3.68e+02]
[ 2.96e+02  3.32e+02  3.68e+02  4.04e+02]
>>> print(pic.tools.lowtri(X)) #doctest: +NORMALIZE_WHITESPACE
[ 2.24e+02]
[ 2.48e+02]
[ 2.72e+02]
```

(continues on next page)

(continued from previous page)

```
[ 2.96e+02]
[ 2.76e+02]
[ 3.04e+02]
[ 3.32e+02]
[ 3.36e+02]
[ 3.68e+02]
[ 4.04e+02]
```

picos.tools.**lse**(*exp*)

A shorthand for *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
<LSE Constraint: logsumexp(A·x) ≤ 0>
```

picos.tools.**ltrim1**(*vec*, *uptri=True*, *offdiag_fact=1.0*)

If `vec` is a vector or an affine expression of size n(n+1)/2, ltrim1(vec) returns a (n,n) matrix with the elements of vec in the lower triangle. If `uptri == False`, the upper triangle is 0, otherwise the upper triangle is the symmetric of the lower one.

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 picos as pic
>>> 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']
<2×1 Affine Expression: [2×3]·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']
<2×1 Affine Expression: B[matrix]·x + [B[foo]]>
```

picos.tools.**norm**(*exp*, *num=2*, *denom=1*)

returns a *NormP_Exp* object representing the (generalized-) p-norm of the entries of `exp[:]`. This can be used to enter constraints of the form $\|x\|_p \leq t$ with $p \geq 1$. Generalized norms are also defined for $p < 1$, by using the usual formula $\mathrm{norm}(x,p) := \left(\sum_i x_i^p\right)^{1/p}$. Note that this function is concave (for $p < 1$) over the set of vectors with nonnegative coordinates. When a constraint of the form $\mathrm{norm}(x,p) > t$ with $p \leq 1$ is entered, PICOS implicitely assumes that x is a nonnegative vector.

This function can also be used to represent the Lp,q- norm of a matrix (for $p, q \geq 1$): $\mathrm{norm}(X,(p,q)) := \left(\sum_i (\sum_j x_{ij}^q)^{p/q}\right)^{1/p}$, that is, the p-norm of the vector formed with the q-norms of the rows of X.

The exponent p of the norm must be specified either by a couple numerator (2d argument) / denominator (3d arguments), or directly by a float `p` given as second argument. In the latter case a rational approximation of `p` will be used. It is also possible to pass `'inf'` as second argument for the infinity-norm (aka max-norm).

For the case of (p,q)-norms, `p` and `q` must be specified by a tuple of floats in the second argument (rational approximations will be used), and the third argument will be ignored.

Example:

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x',1)
>>> y = P.add_variable('y',3)
>>> pic.norm(y,7,3) < x
<p-Norm Constraint: ‖y‖_(7/3) ≤ x>
>>> pic.norm(y,-0.4) > x
<Generalized p-Norm Constraint: ‖y‖_(-2/5) ≥ x>
>>> X = P.add_variable('X',(3,2))
>>> pic.norm(X,(1,2)) < 1
<(p,q)-Norm Constraint: ‖X‖_1,2 ≤ 1>
>>> pic.norm(X,('inf',1)) < 1
<(p,q)-Norm Constraint: ‖X‖_inf,1 ≤ 1>
```

picos.tools.**offset_in_lil**(*lil*, *offset*, *lower*)

substract the `offset` from all elements of the (recursive) list of lists `lil` which are larger than `lower`.

picos.tools.**parameterized_string**(*strings*, *replace='\\d+'*, *placeholders='ijklpqr'*, *fallback='?'*)

Given a list of strings with similar structure, finds a single string with placeholders and an expression that denotes how to instantiate the placeholders in order to obtain each string in the list.

The function is designed to take a number of symbolic string representations of math expressions that differ only with respect to indices.

Parameters

- **strings** (*list*) – The list of strings to compare.
- **replace** (*str*) – A regular expression describing the bits to replace with placeholders.
- **placeholders** – An iterable of placeholder strings. Usually a string, so that each of its characters becomes a placeholder.
- **fallback** (*str*) – A fallback placeholder string, if the given placeholders are not sufficient.

Returns A tuple of two strings, the first being the template string and the second being a description of the placeholders used.

Example

```
>>> from picos.tools import parameterized_string as ps
>>> ps(["A[{}]".format(i) for i in range(5, 31)])
```

(continues on next page)

(continued from previous page)

```
('A[i+5]', 'i ∈ [0...25]')
>>> ps(["A[{}]".format(i) for i in range(5, 31, 5)])
('A[5(i+1)]', 'i ∈ [0...5]')
>>> S=["A[0]·B[2]·C[3]·D[5]·F[0]",
...     "A[1]·B[1]·C[6]·D[6]·F[0]",
...     "A[2]·B[0]·C[9]·D[9]·F[0]"]
>>> ps(S)
('A[i]·B[-(i-2)]·C[3(i+1)]·D[j]·F[0]', '(i,j) ∈ zip([0...2],[5,6,9])')
```

picos.tools.**partial_trace**(*X*, *k=1*, *dim=None*)

Partial trace of an Affine Expression, with respect to the `k` th subsystem for a tensor product of dimensions `dim`. If `X` is a matrix *AffinExp* that can be written as $X = A_0 \otimes \cdots \otimes A_{n-1}$ for some matrices $A_0, \ldots, A_{n-1}$ of respective sizes `dim[0] x dim[0]`, ... , `dim[n-1] x dim[n-1]` (`dim` is a list of ints if all matrices are square), or `dim[0][0] x dim[0][1]`, ...,‘‘dim[n-1][0] x dim[n-1][1]‘‘ (`dim` is a list of 2-tuples if any of them except the `k` th one is rectangular), this function returns the matrix $Y = \operatorname{trace}(A_k) \quad A_0 \otimes \cdots A_{k-1} \otimes A_{k+1} \otimes \cdots \otimes A_{n-1}$.

The default value `dim=None` automatically computes the size of the subblocks, assuming that `X` is a $n^2 \times n^2$-square matrix with blocks of size $n \times n$.

Example:

```
>>> import picos as pic
>>> import cvxopt as cvx
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,4))
>>> X.value = cvx.matrix(range(16),(4,4))
>>> print(X) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
[ 1.00e+00  5.00e+00  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  1.00e+01  1.40e+01]
[ 3.00e+00  7.00e+00  1.10e+01  1.50e+01]
>>> # Partial trace with respect to second subsystem (k=1):
>>> print(pic.partial_trace(X)) #doctest: +NORMALIZE_WHITESPACE
[ 5.00e+00  2.10e+01]
[ 9.00e+00  2.50e+01]
>>> # And with respect to first subsystem (k=0):
>>> print(pic.partial_trace(X,0)) #doctest: +NORMALIZE_WHITESPACE
[ 1.00e+01  1.80e+01]
[ 1.20e+01  2.00e+01]
```

picos.tools.**partial_transpose**(*exp*, *dims_1=None*, *subsystems=None*, *dims_2=None*)

Partial transpose of an Affine Expression, with respect to given subsystems. If `X` is a matrix *AffinExp* that can be written as $X = A_0 \otimes \cdots \otimes A_{n-1}$ for some matrices $A_0, \ldots, A_{n-1}$ of respective sizes `dims_1[0] x dims_2[0]`, ... , `dims_1[n-1] x dims_2[n-1]`, this function returns the matrix $Y = B_0 \otimes \cdots \otimes B_{n-1}$, where $B_i = A_i^T$ if `i in subsystems`, and $B_i = A_i$ otherwise.

The optional parameters `dims_1` and `dims_2` are tuples specifying the dimension of each subsystem A_i. The argument `subsystems` must be a `tuple` (or an `int`) with the index of all subsystems to be transposed.

The default value `dims_1=None` automatically computes the size of the subblocks, assuming that `exp` is a $n^k \times n^k$-square matrix, for the *smallest* appropriate value of $k \in [2, 6]$, that is `dims_1=(n,)*k`.

If `dims_2` is not specified, it is assumed that the subsystems A_i are square, i.e., `dims_2=dims_1`. If `subsystems` is not specified, the default assumes that only the last system must be transposed, i.e., `subsystems = (len(dims_1)-1,)`

Example:

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

(continues on next page)

(continued from previous page)

```
>>> P = pic.Problem()
>>> X = P.add_variable('X',(4,4))
>>> X.value = cvx.matrix(range(16),(4,4))
>>> print(X) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
[ 1.00e+00  5.00e+00  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  1.00e+01  1.40e+01]
[ 3.00e+00  7.00e+00  1.10e+01  1.50e+01]
>>> # Standard partial transpose with respect to the 2x2 blocks and 2nd␣
↪subsystem:
>>> print(X.Tx) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  1.00e+00  8.00e+00  9.00e+00]
[ 4.00e+00  5.00e+00  1.20e+01  1.30e+01]
[ 2.00e+00  3.00e+00  1.00e+01  1.10e+01]
[ 6.00e+00  7.00e+00  1.40e+01  1.50e+01]
>>> # Now with respect to the first subsystem:
>>> print(pic.partial_transpose(X,(2,2),0)) #doctest: +NORMALIZE_WHITESPACE
[ 0.00e+00  4.00e+00  2.00e+00  6.00e+00]
[ 1.00e+00  5.00e+00  3.00e+00  7.00e+00]
[ 8.00e+00  1.20e+01  1.00e+01  1.40e+01]
[ 9.00e+00  1.30e+01  1.10e+01  1.50e+01]
```

picos.tools.**quad2norm**(*qd*)
transform the list of bilinear terms qd in an equivalent squared norm (x.T Q x) -> ||Q**0.5 x||**2

picos.tools.**remove_in_lil**(*lil*, *elem*)
remove the element `elem` from a (recursive) list of list `lil`. empty lists are removed if any

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>, '[3×1]')
>>> 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]) #doctest: +NORMALIZE_
↪WHITESPACE
[   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>, '5.3·I')
```

picos.tools.**simplex**(*gamma=1*)

returns a *TruncatedSimplex* object representing the set $\{x \geq 0 : ||x||_1 \leq \gamma\}$.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.simplex()
<Standard Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 1}>
>>> x << pic.simplex(2)
<Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 2}>
```

picos.tools.**spmatrix**(**args*, ***kwargs*)

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

This is a replacement for Python's *sum* that produces sensible string representations when summing PICOS expressions.

Parameters

- **lst** – A list of *expressions*.
- **it** – DEPRECATED
- **indices** – DEPRECATED

Example:

```
>>> import picos
>>> P = picos.Problem()
>>> x = P.add_variable("x", 5)
>>> e = [x[i]*x[i+1] for i in range(len(x) - 1)]
>>> sum(e)
<Quadratic Expression: x[0]·x[1] + x[1]·x[2] + x[2]·x[3] + x[3]·x[4]>
>>> picos.sum(e)
<Quadratic Expression: ∑(x[i]·x[i+1] : i ∈ [0...3])>
```

picos.tools.**sum_k_largest**(*exp*, *k*)

returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest elements of an affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} x_i^{\downarrow} \leq t$. This kind of constraints is reformulated internally as a set of linear inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',3)
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_largest(x,2) < 1
<Sum of Largest Elements Constraint: sum_2_largest(x) ≤ 1>
>>> pic.sum_k_largest(x,1) < t
<Sum of Largest Elements Constraint: max(x) ≤ t>
```

picos.tools.**sum_k_largest_lambda**(*exp*, *k*)

returns a *Sum_k_Largest_Exp* object representing the sum of the `k` largest eigenvalues of a square matrix affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} \lambda_i^{\downarrow}(X) \leq t$. This kind of constraints is reformulated internally as a set of linear matrix inequalities (SDP). Note that `exp` is assumed to be symmetric (picos does not check).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_largest_lambda(X,3) < 1
<Sum of Largest Eigenvalues Constraint: trace(X) ≤ 1>
>>> pic.sum_k_largest_lambda(X,2) < t
<Sum of Largest Eigenvalues Constraint: sum_2_largest_λ(X) ≤ t>
```

picos.tools.**sum_k_smallest**(*exp*, *k*)

returns a *Sum_k_Smallest_Exp* object representing the sum of the `k` smallest elements of an affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} x_i^{\uparrow} \geq t$. This kind of constraints is reformulated internally as a set of linear inequalities.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> x = prob.add_variable('x',3)
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_smallest(x,2) > t
<Sum of Smallest Elements Constraint: sum_2_smallest(x) ≥ t>
>>> pic.sum_k_smallest(x,1) > 3
<Sum of Smallest Elements Constraint: min(x) ≥ 3>
```

picos.tools.**sum_k_smallest_lambda**(*exp*, *k*)

returns a *Sum_k_Smallest_Exp* object representing the sum of the `k` smallest eigenvalues of a square matrix affine expression `exp`. This can be used to enter constraints of the form $\sum_{i=1}^{k} \lambda_i^{\uparrow}(X) \geq t$. This kind of constraints is reformulated internally as a set of linear matrix inequalities (SDP). Note that `exp` is assumed to be symmetric (picos does not check).

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.sum_k_smallest_lambda(X,1) > 1
<Sum of Smallest Eigenvalues Constraint: λ_min(X) ≥ 1>
>>> pic.sum_k_smallest_lambda(X,2) > t
<Sum of Smallest Eigenvalues Constraint: sum_2_smallest_λ(X) ≥ t>
```

picos.tools.**sumexp**(*x*, *y=None*)

A shorthand for *SumExponential*.

If the second optional argument is passed, the resulting expression is the sum of perspectives of exponentials $\sum_i y_i \exp(x_i/y_i)$, otherwise it is a sum of exponentials $\sum_i \exp(x_i)$.

picos.tools.**svec**(*mat*, *ignore_sym=False*)
returns the svec representation of the cvx matrix `mat`. (see Dattorro, ch.2.2.2.1)

If `ignore_sym = False` (default), the function raises an Exception if `mat` is not symmetric. Otherwise, elements in the lower triangle of `mat` are simply ignored.

picos.tools.**svecm1**(*vec*, *triu=False*)

picos.tools.**svecm1_identity**(*vtype*, *size*)
row wise svec-1 transformation of the identity matrix of size size[0]*size[1]

picos.tools.**svecm1_identity_factor**(*factors*, *variable*)

Returns Whether the variable coefficients are the svec-1 transformation of the identity.

picos.tools.**trace**(*exp*)
trace of a square AffinExp

picos.tools.**tracepow**(*exp*, *num=1*, *denom=1*, *coef=None*)
Returns a *TracePow_Exp* object representing the trace of the pth-power of the symmetric matrix `exp`, where `exp` is an *AffinExp* which we denote by X. This can be used to enter constraints of the form $\mathrm{trace}\, X^p \leq t$ with $p \geq 1$ or $p < 0$, or $\mathrm{trace}\, X^p \geq t$ with $0 \leq p \leq 1$. Note that X is forced to be positive semidefinite when a constraint of this form is entered in PICOS.

It is also possible to specify a `coef` matrix (M) of the same size as `exp`, in order to represent the expression $\mathrm{trace}(MX^p)$. The constraint $\mathrm{trace}(MX^p) \geq t$ can be reformulated with SDP constraints if M is positive semidefinite and $0 < p < 1$.

Trace of power inequalities are internally reformulated as a set of Linear Matrix Inequalities (SDP), or second order cone inequalities if `exp` is a scalar.

The exponent p of the norm must be specified either by a couple numerator (2d argument) / denominator (3d arguments), or directly by a float `p` given as second argument. In the latter case a rational approximation of `p` will be used.

Example:

```
>>> import picos as pic
>>> prob = pic.Problem()
>>> X = prob.add_variable('X',(3,3),'symmetric')
>>> t = prob.add_variable('t',1)
>>> pic.tracepow(X,7,3) < t
<Trace of Power Constraint: trace(X^(7/3)) ≤ t>
>>> pic.tracepow(X,0.6) > t
<Trace of Power Constraint: trace(X^(3/5)) ≥ t>
>>> import cvxopt as cvx
>>> A = cvx.normal(3,3);A=A*A.T # A random semidefinite positive matrix
>>> A = pic.new_param('A',A)
>>> pic.tracepow(X,0.25,coef=A) > t
<Trace of Power Constraint: trace(A·X^(1/4)) ≥ t>
```

picos.tools.**truncated_simplex**(*gamma=1*, *sym=False*)
returns a *TruncatedSimplex* object representing the set:

- $\{x \geq 0 : ||x||_\infty \leq 1,\ ||x||_1 \leq \gamma\}$ if `sym=False` (default)
- $\{x : ||x||_\infty \leq 1,\ ||x||_1 \leq \gamma\}$ if `sym=True`.

Example

```
>>> import picos as pic
>>> P = pic.Problem()
>>> x = P.add_variable('x', 3)
>>> x << pic.truncated_simplex(2)
```

(continues on next page)

(continued from previous page)

```
<Truncated Simplex Constraint: x ∈ {0 ≤ x ≤ 1 : ∑(x) ≤ 2}>
>>> x << pic.truncated_simplex(2,sym=True)
<Symmetrized Truncated Simplex Constraint: x ∈ {-1 ≤ x ≤ 1 : ∑(|x|) ≤ 2}>
```

picos.tools.**utri**(*mat*)

return elements of the (strict) upper triangular part of a cvxopt matrix

CHAPTER 3

Contribution Guide

3.1 Filing a bug report or feature request

3.1.1 Via GitLab

If you have a GitLab account, just head to PICOS' official issue tracker.

3.1.2 Via mail

If you don't have a GitLab account you can still create an issue by writing a mail to `incoming+picos-api/picos@incoming.gitlab.com`. Unlike issues created directly on GitLab, issues created by mail are *not* publicly visible.

3.2 Submitting a code change

The canoncial way to submit a code change is to

1. fork the PICOS repository on GitLab,
2. clone your fork and make your application use it instead of your system's PICOS installation,
3. optionally create a local topic branch to work with,
4. modify the source and commit your changes, and lastly
5. make a pull request on GitLab so that we can test and merge your changes.

If you don't want to create a GitLab account, we are also happy to receive your changes via mail as a patch created by `git patch`.

3.3 Implementing your solver

If you want to implement support for a new solver, all you have to do is update solvers.py where applicable, and add a file called `solver_<name>.py` in the same directory with your implementation. We recommend that you read two or three of the existing solver implementations to get an idea how things are done. If you want to know exactly how PICOS communicates with your implementation, refer to the solver base class in solver.py.

3.4 Implementing a test case

Production and unit test sets are implemented in the files in the tests folder that start with `ptest_` and `utest_`, respectively. If you want to add to our test pool, feel free to either extend these files or create a new set, whatever is appropriate. Make sure that the tests you add are not too computationally expensive since they are also run as part of our continuous integration pipeline whenever a commit is pushed to GitLab.

3.5 Coding guidelines

3.5.1 Cleanup in progress

We are aiming to tidy up PICOS' codebase in the future to make it more robust and easier to maintain and extend. That means that the cost of adding a new feature often is to also refactor around the neighboring features. That being said, you are encouraged to just rewrite a function that you feel does not look so good, even if you initially just planned to add some text to it! :wink:

3.5.2 Test coverage

Refactoring means stability in the long run, but can break features in the short term. To prevent this from happening, we're happy to grow our set of production and unit test cases. Hence, if you are running a local test to see if your changes work, consider adding it as a permanent test case so that your feature is protected from our clumsiness in the future.

CHAPTER 4

Changelog

This file documents major changes to PICOS. The format is based on Keep a Changelog.

4.1 Unreleased

4.1.1 Added

- Support for ECOS 2.0.7.

4.2 1.2.0 - 2019-01-11

4.2.1 Important

- *A scalar expression's value* and *a scalar constraint's dual* are returned as scalar types as opposed to 1×1 matrices.
- The dual value returned for rotated second order cone constraints is now a proper member of the dual cone (which equals the primal cone up to a factor of 4). Previously, the dual of an equivalent second order cone constraint was returned.
- The Python 2/3 compatibility library `six` is no longer a dependency.

4.2.2 Added

- Support for the ECOS solver.
- Experimental support for MOSEK's new Fusion API.
- Full support for exponential cone programming.
- A production testing framework featuring around 40 novel optimization test cases that allows quick selection of tests, solvers, and solver options.
- A "glyph" system that allows the user to adjust the string representations of future expressions and constraints. For instance, *picos.latin1()* disables use of unicode symbols.
- Support for symmetric variables with all solvers, even if they do not support semidefinite programming.

4.2.3 Changed

- Solver implementations each have a source file of their own, and a common interface that makes implementing new solvers easier.
- Likewise, constraint implementations each have a source file of their own.
- The implementations of CPLEX, Gurobi, MOSEK and SCIP have been rewritten.
- Solver selection takes into account how well a problem is supported, distinguishing between native, secondary, experimental and limited support.
- Unsupported operations on expressions now produce meaningful exceptions.
- *add_constraint* and *add_list_of_constraints* always return the constraints passed to them.
- *add_list_of_constraints* and *picos.sum* find a short string representation automatically.

4.2.4 Removed

- The old production testing script.
- Support for the SDPA solver.
- Support for sequential quadratic programming.
- The options `convert_quad_to_socp_if_needed`, `pass_simple_cons_as_bound`, `return_constraints`, `handleBarVars`, `handleConeVars` and `smcp_feas`.
- Support for GLPK and MOSEK through CVXOPT.

4.2.5 Fixed

- Performance issues when exporting variable bounds to CVXOPT.
- Hadamard product involving complex matrices.
- Adding constant terms to quadratic expression.
- Incorrect or redundant expression string representations.
- GLPK handling of the default `maxit` option.
- Miscellaneous solver-specific bugs in the solvers that were re-implemented.

4.3 1.1.3 - 2018-10-05

4.3.1 Added

- Support for the solvers GLPK and SCIP.
- PICOS packages on Anaconda Cloud.
- PICOS packages in the Arch Linux User Repository.

4.3.2 Changed

- The main repository has moved to GitLab.
- Releases of packages and documentation changes are automated and thus more frequent. In particular, post release versions are available.
- Test bench execution is automated for greater code stability.
- Improved test bench output.
- Improved support for the SDPA solver.
- *partial_trace* can handle rectangular subsystems.

- The documentation was restructured; examples were converted to Python 3.

4.3.3 Fixed

- Upper bounding the norm of a complex scalar.
- Multiplication with a complex scalar.
- A couple of Python 3 specific errors, in particular when deleting constraints.
- All documentation examples are reproducible with the current state of PICOS.

4.4 1.1.2 - 2016-07-04

4.4.1 Added

- Ability to dynamically add and remove constraints, see *the documentation on constraint deletion*.
- Option `pass_simple_cons_as_bound`, see below.

4.4.2 Changed

- *`add_constraint`* returns a handle to the constraint when the option `return_constraints` is set.
- New signature for the function *`partial_transpose`*, which can now transpose arbitrary subsystems from a kronecker product.
- PICOS no longer turns constraints into variable bounds, unless the new option `pass_simple_cons_as_bound` is enabled.

4.4.3 Fixed

- Minor bugs with complex expressions.

4.5 1.1.1 - 2015-08-29

4.5.1 Added

- Support for the SDPA solver.
- Partial trace of an affine expression, see *`partial_trace`*.

4.5.2 Changed

- Improved PEP 8 compliance.

4.5.3 Fixed

- Compatibility with Python 3.

4.6 1.1.0 - 2015-04-15

4.6.1 Added

- Compatibility with Python 3.

4.6.2 Changed

- The main repository has moved to GitHub.

4.7 1.0.2 - 2015-01-30

4.7.1 Added

- Ability to read and write problems in conic benchmark format.
- Support for inequalities involving the sum of the k largest or smallest elements of an affine expression, see *sum_k_largest* and *sum_k_smallest*.
- Support for inequalities involving the sum of the k largest or smallest eigenvalues of a symmetric matrix, see *sum_k_largest_lambda*, *sum_k_smallest_lambda*, *lambda_max* and *lambda_min*.
- Support for inequalities involving the $L_{p,q}$-norm of an affine expression, see *norm*.
- Support for equalities involving complex coefficients.
- New *variable type* for antisymmetric matrix variables: `antisym`.
- Set expressions that affine expressions can be constrained to be an element of, see *ball*, *simplex* and *truncated_simplex*.
- Shorthand functions *maximize* and *minimize* to specify the objective function of a problem and solve it.
- Hadamard (elementwise) product of affine expression, as an overload of the ^ operator, read *the tutorial on overloads*.
- Partial transposition of an aAffine Expression, see *partial_transpose* and *AffinExp.Tx*.

4.7.2 Changed

- Improved efficiency of the sparse SDPA file format writer.
- Improved efficiency of *to_real*.

4.7.3 Fixed

- Scalar product of hermitian matrices.
- Conjugate of a complex expression.

4.8 1.0.1 - 2014-08-27

4.8.1 Added

- Support for semidefinite programming over the complex domain, see *the documentation on complex problems*.
- Helper function to input (multicommodity) graph flow problems, see *the tutorial on flow constraints*.
- Additional `coef` argument to *tracepow*, to represent constraints of the form $\mathrm{trace}(MX^p) \geq t$.

4.8.2 Changed

- Significantly improved slicing performance for *affine expressions*.
- Improved performance of *retrieve_matrix*.
- Improved performance when retrieving primal solution from CPLEX.
- The documentation received an overhaul.

4.9 1.0.0 - 2013-07-19

4.9.1 Added

- Ability to express rational powers of affine expressions with the `**` operator, traces of matrix powers with *picos.tracepow*, (generalized) p-norms with *picos.norm* and n-th roots of a determinant with *picos.detrootn*.
- Ability to specify variable bounds directly rather than by adding constraints, see *add_variable*, *set_lower()*, *set_upper()*, *set_sparse_lower()* and *set_sparse_upper()*.
- Problem dualization, see *dualize*.
- Option `solve_via_dual` which controls passing the dual problem to the solver instead of the primal problem. This can result in a significant speedup for certain problems.
- Semidefinite programming interface for MOSEK 7.0.
- Options `handleBarVars` and `handleConeVars` to customize how SOCPs and SDPs are passed to MOSEK. When these are set to `True`, PICOS tries to minimize the number of variables of the MOSEK instance.

4.9.2 Changed

- If the chosen solver supports this, updated problems will be partially re-solved instead of solved from scratch.

4.9.3 Removed

- Option `onlyChangeObjective`.

4.10 0.1.3 - 2013-04-17

4.10.1 Added

- A *geomean* function to construct geometric mean inequalities that will be cast as SOCP constraints.
- Options `uboundlimit` and `lboundlimit` to tell CPLEX to stop the search as soon as the given threshold is reached for the upper and lower bound, respectively.
- Option `boundMonitor` to inspect the evolution of CPLEX lower and upper bounds.
- Ability to use the weak inequality operators as an alias for the strong ones.

4.10.2 Changed

- The solver search time is returned in the dictionary returned by *solve*.

4.10.3 Fixed

- Access to dual values of fixed variables with CPLEX.
- Evaluation of constant affine expressions with a zero coefficient.
- Number of constraints not being updated in *remove_constraint*.

4.11 0.1.2 - 2013-01-10

4.11.1 Fixed

- Writing SDPA files. The lower triangular part of the constraint matrix was written instead of the upper triangular part.

- A wrongly raised `IndexError` from *`remove_constraint`*.

4.12 0.1.1 - 2012-12-08

4.12.1 Added

- Interface to Gurobi.
- Ability to give an initial solution to warm-start mixed integer optimizers.
- Ability to get a reference to a constraint that was added.

4.12.2 Fixed

- Minor bugs with quadratic expressions.

4.13 0.1.0 - 2012-06-22

4.13.1 Added

- Initial release of PICOS.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z