PICOS Documentation

Release 2.5

Guillaume Sagnol Maximilian Stahlberg

2024-11-04

CONTENTS

CHAPTER

ONE

INTRODUCTION

PICOS is a user friendly Python API to several conic and integer programming solvers, designed to be used by both application developers and researchers as well as instructors teaching courses on mathematical optimization. It allows you to enter an optimization problem as a **high level model**, with painless support for **(complex) vector and matrix variables** and **multidimensional 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 supports the following solvers and problem types. To use a solver, you need to separately install it along with the Python interface listed here.

Solver	Python interface	LP	SOCP, QCQP	SDP	EXP	MIP	QREP	License
CPLEX	included	Yes	Yes			Yes		non-free
CVX-OPT	native	Yes	Yes	Yes	GP			GPL-3
ECOS	ecos-python	Yes	Yes		Yes	Yes		GPL-3
GLPK	swiglpk	Yes				Yes		GPL-3
Gurobi	gurobipy	Yes	Yes			Yes		non-free
MOSEK	included	Yes	Yes	Yes		Yes		non-free
OSQP	native	Yes	QP					Apache-2.0
QICS	native	Yes	Yes	Yes	Yes		Yes	MIT
SCIP	PySCIPOpt	Yes	Yes			Yes		ZIB/MIT
SMCP	native			Yes				GPL-3

Example

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

```
>>> import picos as pc
>>> P = pc.Problem()
>>> x = pc.IntegerVariable("x", 2)
>>> P += 2*x <= 11
>>> P.maximize = pc.sum(x)
>>> P.solve(solver="glpk")  # Optional: Use GLPK as backend.
<feasible primal solution (claimed optimal) from glpk>
>>> P.value
10.0
>>> print(x)
[ 5.00e+00]
[ 5.00e+00]
```

You can head to our quick examples or the tutorial for more.

1.2 Installation

As of release 2.2, PICOS requires **Python 3.4** or later.

Via pip

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

Via Anaconda

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

Via your system's package manager

Distribution	Latest major version	Latest version
Arch Linux	python-picos	python-picos-git

If you are packaging PICOS for additional platforms, please let us know.

From source

The PICOS source code can be found on GitLab. There are only two dependencies:

- NumPy
- CVXOPT

1.3 Documentation

The full documentation can be browsed online or downloaded in PDF form.

1.4 Credits

Developers

- Guillaume Sagnol has started work on PICOS in 2012.
- Maximilian Stahlberg is extending and co-maintaining PICOS since 2017.

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, see also the old contributors page on GitHub.

1.5 Citing

The preferred way to cite PICOS in your research is our JOSS paper:

```
@article{PICOS,
  author  = {Guillaume Sagnol and Maximilian Stahlberg},
  journal = {Journal of Open Source Software},
  title   = {{PICOS}: A {Python} interface to conic optimization solvers},
  year    = {2022},
  issn    = {2475-9066},
  month   = feb,
  number  = {70},
  pages   = {3915},
  volume  = {7},
  doi     = {10.21105/joss.03915},
}
```

If citing a specific version of PICOS is necessary, then we offer also source deposits on Zenodo.

1.6 License

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

CHAPTER

TWO

TUTORIAL

First of all, let us import PICOS:

```
>>> import picos
```

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 *ascii* or *latin1* to restrict the charset used:

```
>>> X = picos.SymmetricVariable("X", 4)       # Create a dummy variable.
>>> print(X >> 0)                             # Default representation of X ⪰ 0.
X ⪰ 0
>>> picos.latin1()                            # Limit to ISO 8859-1 (Latin-1).
>>> print(X >> 0)
X » 0
>>> picos.ascii()                             # Limit to pure ASCII.
>>> print(X >> 0)
X >> 0
```

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

```
>>> picos.default_charset()  # The same as picos.unicode().
```

2.1 Variables

Every optimization endeavor starts with variables. As of PICOS 2.0, the preferred way to create variables is to create an instance of the desired variable class:

```
>>> from picos import RealVariable, BinaryVariable
>>> t = RealVariable("t")                     # A scalar.
>>> x = RealVariable("x", 4)                  # A column vector with 4 elements.
>>> Y = RealVariable("Y", (2, 4))             # A 2×4 matrix.
>>> Z = RealVariable("Z", (4, 2))             # A 4×2 matrix.
>>> w = BinaryVariable("w")                   # A binary scalar.
```

Now, let's inspect these variables:

```
>>> w
<1×1 Binary Variable: w>
>>> Y
<2×4 Real Variable: Y>
>>> x.shape
```

(continues on next page)

(continued from previous page)

```
(4, 1)
>>> Z.name
'Z'
```

Assigning a value

Assigning values to variables is usually the solver's job, but we can do it manually:

```
>>> t.value = 2
>>> # In the case of a binary variable, we can only assign a (near) 0 or 1:
>>> w.value = 0.5
Traceback (most recent call last):
  ...
ValueError: Failed to assign a value to mutable w: Data is not near-binary with
    absolute tolerance 1.0e-04: Largest difference is 5.0e-01.
>>> print(Z)
Z
>>> Z.value = range(8)
>>> print(Z)  # If a variable is valued, prints the value instead.
[ 0.00e+00  4.00e+00]
[ 1.00e+00  5.00e+00]
[ 2.00e+00  6.00e+00]
[ 3.00e+00  7.00e+00]
```

As you can see from the last example, PICOS uses column-major order when loading one-dimensional data such as a Python `range` into a matrix. The documentation of *`load_data`* explains PICOS' data loading concept in greater detail.

2.2 Affine expressions

The fundamental building blocks of optimization models are affine (matrix) expressions. Each entry of such an expression is simply a linear combination of any number of scalar variables plus a constant offset. The variable objects that we have defined above are special cases of affine expression that refer to themselves via an identity transformation.

We can now use our variables to create more advanced affine expressions, which are stored as instances of *`ComplexAffineExpression`* or of its subclass *`AffineExpression`*. For instance, we may transpose a matrix variable using the suffix `.T`:

```
>>> Y
<2×4 Real Variable: Y>
>>> Y.T
<4×2 Real Linear Expression: Yᵀ>
```

PICOS expression types overload the standard Python math operators so that you can denote, for instance, the sum of two expressions as follows:

```
>>> Z + Y.T
<4×2 Real Linear Expression: Z + Yᵀ>
```

The overloaded operators will convert arbitrary data on the fly:

```
>>> t + 1
<1×1 Real Affine Expression: t + 1>
```

(continues on next page)

(continued from previous page)

```
>>> x + 1  # The 1 is broadcasted to a 4×1 vector of all ones.
<4×1 Real Affine Expression: x + [1]>
```

Constants

Constants are simply affine expressions with no linear part and are more commonly referred to as *data*. By default, PICOS uses a short dummy string to represent multidimensional constants, and reshapes them as needed:

```
>>> Y + [1, -2, 3, -4, 5, -6, 7, 8]          # Load list as a 2×4 matrix.
<2×4 Real Affine Expression: Y + [2×4]>
```

If you want to give your constant data a meaningful name and fix its shape for more type safety, you can do this using `Constant`:

```
>>> from picos import Constant
>>> alpha = Constant("α", 23)                 # Load 23 under the name α.
>>> b = Constant("b", range(4))               # Load as a column vector.
>>> C = Constant("C", [1, -2, 3, -4, 5, -6, 7, 8], (2, 4)); C
<2×4 Real Constant: C>
>>> Y + C
<2×4 Real Affine Expression: Y + C>
```

The data loading semantics of `Constant` or when loading data on the fly are the same as when valuing variables (`load_data`). In particular, you can seamlessly input CVXOPT or NumPy matrices:

```
>>> import numpy
>>> Y + numpy.array([[1, 2, 3, 4], [5, 6, 7, 8]])
<2×4 Real Affine Expression: Y + [2×4]>
```

2.3 Overloaded operators

Now that we have some variables (t, x, w, Y and Z) and a couple of constant parameters (α, b, C), let us create some more affine expressions with them:

```
>>> C.shape, Z.shape                          # Recall the shapes.
((2, 4), (4, 2))
>>> C*Z                                       # Left multiplication.
<2×2 Real Linear Expression: C·Z>
>>> Z*C                                       # Right multiplication.
<4×4 Real Linear Expression: Z·C>
>>> C*Z*C                                     # Left and right multiplication.
<2×4 Real Linear Expression: C·Z·C>
>>> alpha*Y                                   # Scalar multiplication.
<2×4 Real Linear Expression: α·Y>
>>> t/alpha - alpha/2                         # Division and subtraction.
<1×1 Real Affine Expression: t/α - α/2>
>>> (b | x)                                   # Dot product.
<1×1 Real Linear Expression: ⟨b, x⟩>
>>> # Generalized dot product for matrices: ⟨A, B⟩ = tr(Aᴴ·B):
>>> (C | Y)
<1×1 Real Linear Expression: ⟨C, Y⟩>
>>> b^x                                       # Hadamard (element-wise) product.
<4×1 Real Linear Expression: b⊙x>
```

(continues on next page)

(continued from previous page)

```
>>> C@Z                                             # Kronecker product.
<8×8 Real Linear Expression: C⊗Z>
```

Slicing

Python slicing notation can be used to extract single elements or submatrices:

```
>>> Y[0, 1]                                         # Element in 1st row, 2nd column.
<1×1 Real Linear Expression: Y[0,1]>
>>> x[1:3]                                          # 2nd and 3rd element of x.
<2×1 Real Linear Expression: x[1:3]>
>>> x[-1]                                           # Last element of x.
<1×1 Real Linear Expression: x[-1]>
>>> Y[1,:]                                          # 2nd row of Y.
<1×4 Real Linear Expression: Y[1,:]>
>>> C[:, 1:3]*Y[:, -2::-2]                          # Extended slicing with step size.
<2×2 Real Linear Expression: C[:,1:3]·Y[:,-2::-2]>
```

In the last example, we select only the second and third column of C as well as the columns of Y with an even index considered in reverse order. The full power and notation of slicing is explained in *Matrix Slicing*.

Concatenation

We can also create larger affine expressions by concatenating them vertically with & or horizontally with //:

```
>>> (b & 2*b & x & C.T*C*x) // x.T
<5×4 Real Affine Expression: [b, 2·b, x, Cᵀ·C·x; xᵀ]>
```

You have to be a little careful when it comes to operator precedence, because Python has the binding strength of & and // built into its grammar with logical disjunction and integral division in mind. When in doubt, use parenthesis around your blocks.

Broadcasting and reshaping

To recall an example we've seen earlier with variables, scalars are broadcasted to the necessary shape to allow an addition or subtraction to take place:

```
>>> 5*x - alpha
<4×1 Real Affine Expression: 5·x - [α]>
```

Note, however, that apart from this simple broadcasting rule, the shape of a PICOS constant (loaded via `Constant`) is already fixed. You can't add a 8×1 vector to a 4×2 matrix:

```
>>> Z + (x // b)
Traceback (most recent call last):
  ...
TypeError: Invalid operation BiaffineExpression.__add__(Z, [x; b]):
    The operand shapes of 4×2 and 8×1 do not match.
```

The reason is simply that PICOS does not know *which* side to reshape. You can make the example work by being more explicit:

```
>>> Z + (x // b).reshaped((4, 2))
<4×2 Real Affine Expression: Z + reshaped([x; b], 4×2)>
```

Summing multiple expressions

Since affine expressions overload +, you could use Python's sum to add a bunch of them. However, the string representation can become rather long:

```
>>> # Create a sequence of matrix constants with sensible names:
>>> A = [Constant("A[{}]".format(i), range(i, i + 8), (2, 4)) for i in range(5)]
>>> A[0]
<2×4 Real Constant: A[0]>
>>> sum([A[i]*Z for i in range(5)])
<2×2 Real Linear Expression: A[0]·Z + A[1]·Z + A[2]·Z + A[3]·Z + A[4]·Z>
```

To obtain a shorter representation, use *picos.sum* instead:

```
>>> picos.sum([A[i]*Z for i in range(5)])
<2×2 Real Linear Expression: ∑(A[i]·Z : i ∈ [0...4])>
```

This works for all kinds of expressions and will look hard to find some pattern in the summands' string descriptions.

2.4 Norms and quadratics

Norms

The norm of an affine expression can be expressed using Python's built-in abs function. If x is an affine vector, abs(x) denotes its Euclidean norm $\sqrt{x^T x}$:

```
>>> abs(x)
<Euclidean Norm: ‖x‖>
```

If the affine expression is a matrix, abs returns its Frobenius norm $\|M\|_F = \sqrt{\operatorname{trace}(M^T M)}$:

```
>>> abs(Z - 2*C.T)
<Frobenius Norm: ‖Z - 2·Cᵀ‖>
```

The absolute value of a scalar is expressed in the same way:

```
>>> abs(t)
<Absolute Value: |t|>
```

As is the modulus of a complex expression:

```
>>> t + 1j
<1×1 Complex Affine Expression: t + 1j>
>>> abs(t + 1j)
<Complex Modulus: |t + 1j|>
```

Additional norms are available through the *Norm* class.

Quadratics

Quadratic expressions can be formed in several ways:

```
>>> abs(x)**2                                 # Squared norm.
<Squared Norm: ‖x‖²>
>>> t**2 - x[1]*x[2] + 2*t - alpha            # Sum involving quadratic terms.
<Quadratic Expression: t² - x[1]·x[2] + 2·t - α>
>>> (x[1] - 2) * (t + 4)                      # Product of affine expressions.
<Quadratic Expression: (x[1] - 2)·(t + 4)>
>>> Y[0,:]*x                                  # Row vector times column vector.
<Quadratic Expression: Y[0,:]·x>
>>> (x + 2 | Z[:,1])                          # Scalar product.
<Quadratic Expression: ⟨x + [2], Z[:,1]⟩>
>>> (t & alpha) * C * x                       # Quadratic form.
<Quadratic Expression: [t, α]·C·x>
```

Note that there is no natural way to define a vector or matrix of quadratic expressions. In PICOS, only affine expressions can be multidimensional.

2.5 Defining a problem

Now that we know how to construct affine and quadratic expressions and norms, it is time to use them as part of an optimization problem:

```
>>> from picos import Problem
>>> P = Problem()
>>> P.set_objective("min", (t - 5)**2 + 2)
>>> print(P)
Quadratic Program
  minimize (t - 5)² + 2
  over
    1×1 real variable t
```

Next we'll search a solution for this problem, but first we configure that only the solver CVXOPT may be used so that the documentation examples are reproducible. We can do this by assigning to the problem's *options* attribute:

```
>>> P.options.solver = "cvxopt"
```

We can now obtain a solution by calling *solve*:

```
>>> solution = P.solve()
>>> solution
<feasible primal solution (claimed optimal) from cvxopt>
>>> solution.primals
{<1×1 Real Variable: t>: [4.999997568104307]}
```

Unless disabled by passing `apply_solution=False` to *solve*, the solution is automatically applied to the variables involved in the problem definition, so that the entire Problem is now valued:

```
>>> round(t, 5)
5.0
>>> round(P, 5)
2.0
```

The Python functions `round`, `int`, `float` and `complex` are automatically applied to the `value` attribute of variables, expressions and problems.

2.6 Setting options

We've already seen the `solver` option used which allows you to take control over which of the available solvers should be used. You can display all available options and their default values by printing the *options* instance (we've cut some from the output):

```
>>> print(P.options)
Modified solver options:
  solver              = cvxopt (default: None)

Default solver options:
  ...
  apply_solution      = True
  ...
  verbosity           = 0
  ...
```

If you want to change an option only for a single solution attempt, you can also pass it to *solve* as a keyword argument:

```
>>> # Solve again but don't apply the result.
>>> solution = P.solve(apply_solution=False)
```

2.7 Constraints

Constrained optimization is only half the fun without the constraints. PICOS again provides overloaded operators to define them:

```
>>> t <= 5
<1×1 Affine Constraint: t ≤ 5>
>>> x[0] == x[-1]
<1×1 Affine Constraint: x[0] = x[-1]>
>>> abs(x)**2 <= t
<Squared Norm Constraint: ‖x‖² ≤ t>
>>> abs(x)**2 >= t
<Nonconvex Quadratic Constraint: ‖x‖² ≥ t>
```

Unless there are solvers or reformulation strategies that can deal with a certain nonconvex constraint type, as is the case for the $\|x\|^2 \geq t$ constranint above, PICOS will raise a `TypeError` to let you know that such a constraint is not supported:

```
>>> abs(x) <= t
<5×1 SOC Constraint: ‖x‖ ≤ t>
>>> abs(x) >= t
Traceback (most recent call last):
  ...
TypeError: Cannot lower-bound a nonconcave norm.
```

When working with multidimensional affine expressions, the inequality operators >= and <= are understood element-wise (or to put it more mathy, they represent conic inequality with respect to the nonnegative orthant):

```
>>> Y >= C
<2×4 Affine Constraint: Y ≥ C>
```

It is possible to define linear matrix inequalities for use in semidefinite programming with the operators >> and << denoting the Loewner order:

```
>>> from picos import SymmetricVariable
>>> S = SymmetricVariable("S", 4)
>>> S >> C.T*C
<4×4 LMI Constraint: S ≽ Cᵀ·C>
```

Other conic inequalities do not have a Python operator of their own, but you can denote set membership of an affine expression in a cone. To make this possible, the operator << is also overloaded to denote "is element of":

```
>>> abs(x) <= t             # Recall that this is a second order cone inequality.
<5×1 SOC Constraint: ‖x‖ ≤ t>
>>> t // x << picos.soc()  # We can also write it like this.
<5×1 SOC Constraint: ‖[t; x][1:]‖ ≤ [t; x][0]>
```

Here *soc* is a shorthand for *SecondOrderCone*, defined as the convex set

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_1 \geq \sqrt{\sum_{i=2}^{n} x_i^2} \right\}.$$

Similarly, we can constrain an expression to be in the rotated second order cone

$$\mathcal{R}_p^n = \left\{ x \in \mathbb{R}^n \;\middle|\; p x_1 x_2 \geq \sum_{i=2}^{n} x_i^2 \wedge x_1, x_2 \geq 0 \right\}$$

parameterized by p:

```
>>> picos.rsoc(p=1) >> x
<4×1 RSOC Constraint: ‖x[2:]‖² ≤ x[0]·x[1] ∨ x[0], x[1] ≥ 0>
```

Other sets you can use like this include *Ball*, *Simplex* and the *ExponentialCone*.

2.8 Constrained optimization

Let's get back to our quadratic program P, which we have already solved to optimality with $t = 5$:

```
>>> print(P)
Quadratic Program
  minimize (t - 5)² + 2
  over
    1×1 real variable t
```

Adding constraints

We can now add the constraints that t must be the sum over all elements of x and that every element of x may be at most 1:

```
>>> Csum = P.add_constraint(t == x.sum)
>>> Cone = P.add_constraint(x <= 1)
>>> print(P)
Quadratic Program
  minimize (t - 5)² + 2
  over
    1×1 real variable t
    4×1 real variable x
  subject to
    t = ∑(x)
    x ≤ [1]
```

Now let's solve the problem again and see what we get:

```
>>> P.solve()
<primal feasible solution pair (claimed optimal) from cvxopt>
>>> round(P, 5)
3.0
>>> round(t, 5)
4.0
>>> x.value
<4x1 matrix, tc='d'>
>>> print(x.value)
[ 1.00e+00]
[ 1.00e+00]
[ 1.00e+00]
[ 1.00e+00]
```

Note that multidimensional values such as that of x are returned as CVXOPT matrix types.

Slack and duals

Since our problem has constraints, we now have slack values and a dual solution as well:

```
>>> Csum.slack
-0.0
>>> Csum.dual
2.000004393989704
>>> print(Cone.slack)
[ 9.31e-12]
[ 9.31e-12]
[ 9.31e-12]
[ 9.31e-12]

>>> print(Cone.dual)
[ 2.00e+00]
[ 2.00e+00]
[ 2.00e+00]
[ 2.00e+00]
```

We did not round the values this time, to showcase that solvers don't always produce exact solutions even if the problem is "easy". The variable t is also not exactly 4:

```
>>> t.value
3.999999999962744
```

To learn more about dual values, see *Dual Values*. For controlling the numeric precision requirements, see *Numeric Tolerances*.

Removing constraints

Let's say we are not happy with our upper bound on x and we'd rather constrain it to be inside a unit simplex. We can remove the former constraint as follows:

```
>>> P.remove_constraint(Cone)
```

Instead of the constraint itself, we could also have supplied its index in the problem, as constraints remain in the order in which you add them. Now let's add the new constraint:

```
>>> Csimplex = P.add_constraint(x << picos.Simplex())
>>> print(P)
Quadratic Program
  minimize (t - 5)² + 2
  over
    1×1 real variable t
    4×1 real variable x
  subject to
    t = ∑(x)
    x ∈ {x ≥ 0 : ∑(x) ≤ 1}
```

If we solve again we expect t to be 1:

```
>>> solution = P.solve()
>>> round(t, 5)
1.0
```

If the selected solver supports this, changes to a problem's constraints and objective are passed in the form of updates to the solver's internal state which can make successive solution searches much faster. Unfortunately, CVXOPT is stateless so we don't get an advantage here.

Grouping constraints

You can also add and remove constraints as a group. Let's compute four real numbers between 0 and 1, represented by x_1 to x_4 (`x[0]` to `x[3]`), such that their minimum distance is maximized:

```
>>> from pprint import pprint
>>> P.reset()                                # Reset the problem, keep options.
>>> d = RealVariable("d", 3)                 # A vector of distances.
>>> P.set_objective("max", picos.min(d))     # Maximize the minimum distance.
>>> C1 = P.add_constraint(x[0] >= 0)         # Numbers start at 0.
>>> C2 = P.add_constraint(x[3] <= 1)         # And end at 1.
>>> # Use constraint groups to order the x[i] and map their distance to y:
>>> G1 = P.add_list_of_constraints([x[i - 1] <= x[i] for i in range(4)])
>>> G2 = P.add_list_of_constraints([d[i] == x[i+1] - x[i] for i in range(3)])
>>> pprint(G1)                               # Show the constraints added.
[<1×1 Affine Constraint: x[-1] ≤ x[0]>,
 <1×1 Affine Constraint: x[0] ≤ x[1]>,
 <1×1 Affine Constraint: x[1] ≤ x[2]>,
 <1×1 Affine Constraint: x[2] ≤ x[3]>]
>>> pprint(G2)
[<1×1 Affine Constraint: d[0] = x[1] - x[0]>,
 <1×1 Affine Constraint: d[1] = x[2] - x[1]>,
 <1×1 Affine Constraint: d[2] = x[3] - x[2]>]
>>> print(P)
Optimization Problem
  maximize min(d)
```

(continues on next page)

(continued from previous page)

```
  over
    3×1 real variable d
    4×1 real variable x
  subject to
    x[0] ≥ 0
    x[3] ≤ 1
    x[i-1] ≤ x[i] ∀ i ∈ [0...3]
    d[i] = x[i+1] - x[i] ∀ i ∈ [0...2]
```

This looks promising and the constraint groups are nicely formatted, let's solve the problem and see what we get:

```
>>> P.solve()
<primal feasible solution pair (claimed optimal) from cvxopt>
>>> print(x)
[ 5.00e-01]
[ 5.00e-01]
[ 5.00e-01]
[ 5.00e-01]
>>> print(d)
[ 1.88e-11]
[ 1.88e-11]
[ 1.88e-11]
```

Apparently there is an error! Revisiting our problem definition, it seems the first constraint in G_1, that is `x[-1] <= x[0]`, was unnecessary and forces all x_i to take the same value. Luckily, we can remove it from the group by first specifying the group to access (counting single constraints as groups of size one) and then the constraint to remove from it:

```
>>> P.remove_constraint((2, 0))         # Remove 1st constraint from 3rd group.
>>> pprint(P.get_constraint((2,)))      # Show the modified 3rd group.
[<1×1 Affine Constraint: x[0] ≤ x[1]>,
 <1×1 Affine Constraint: x[1] ≤ x[2]>,
 <1×1 Affine Constraint: x[2] ≤ x[3]>]
```

Now it should work:

```
>>> print(P)
Optimization Problem
  maximize min(d)
  over
    3×1 real variable d
    4×1 real variable x
  subject to
    x[0] ≥ 0
    x[3] ≤ 1
    x[i] ≤ x[i+1] ∀ i ∈ [0...2]
    d[i] = x[i+1] - x[i] ∀ i ∈ [0...2]
>>> _ = P.solve()  # Don't show or save the solution object.
>>> print(x)
[ ...]
[ 3.33e-01]
[ 6.67e-01]
[ 1.00e+00]
>>> print(d)
[ 3.33e-01]
[ 3.33e-01]
[ 3.33e-01]
```

(If you see an ellipsis … in an example that means we've cut out a near-zero to allow the other values to be validated automatically.)

CHAPTER

THREE

EXAMPLES

The *quick examples* are all self-contained and can be copy-pasted to a source file or Python console to reproduce them. Most of the remaining examples have a tutorial character and are presented in multiple dependent code sections.

3.1 Quick examples

The short examples below are all self-contained and can be copied to a Python source file or pasted into a Python console.

3.1.1 Projection onto a convex hull

We solve the problem

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & \|Ax - b\| \\ \text{subject to} & \sum_{i=1}^{n} x_i = 1, \\ & x \succeq 0, \end{array}$$

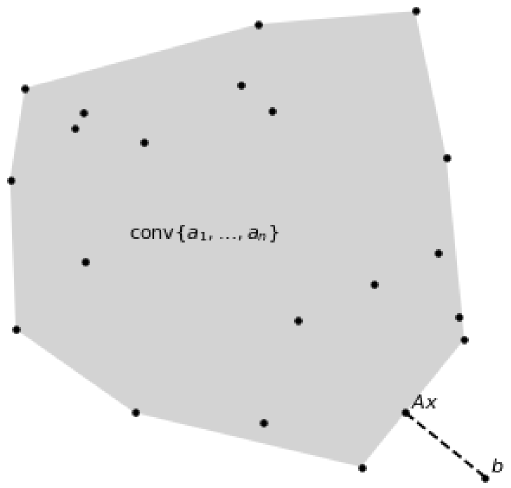
which asks for the projection Ax of the point $b \in \mathbb{R}^m$ onto the convex hull of the columns of $A \in \mathbb{R}^{m \times n}$:

```python
#!/usr/bin/env python3

import numpy as np
import picos as pc
from matplotlib import pyplot
from scipy import spatial

# Make the result reproducible.
np.random.seed(12)

# Define the data.
n = 20
A = np.random.rand(2, n)
b = np.array([1, 0])

# Define the decision variable.
x = pc.RealVariable("x", n)

# Define and solve the problem.
P = pc.Problem()
P.minimize = abs(A*x - b)
```

(continues on next page)

(continued from previous page)

```
P += pc.sum(x) == 1, x >= 0
P.solve(solver="cvxopt")

# Obtain the projection point.
p = (A*x).np

# Plot the results.
V = spatial.ConvexHull(A.T).vertices
figure = pyplot.figure(figsize=(8.7, 4))
figure.gca().set_aspect("equal")
pyplot.axis("off")
pyplot.fill(A.T[V, 0], A.T[V, 1], "lightgray")
pyplot.plot(A.T[:, 0], A.T[:, 1], "k.")
pyplot.plot(*zip(b, p), "k.--")
pyplot.annotate("$\mathrm{conv} \{a_1, \ldots, a_n\}$", [0.25, 0.5])
pyplot.annotate("$b$", b + 1/100)
pyplot.annotate("$Ax$", p + 1/100)
pyplot.tight_layout()
pyplot.show()
```



Example notes

- The Python builtin function `abs` (absolute value) is understood as the default norm. For real vectors, this is the Euclidean norm.
- The attribute *np* returns the value of a PICOS expression as a NumPy type.
- The choice of the CVXOPT solver is optional. Explicit solver choice is made throughout the documentation to make its automatic validation more reliable.

3.1.2 Worst-case projection

We solve the same problem as before but now we assume that the point b to be projected is only known to live inside an ellipsoid around its original location. In this case we cannot hope to obtain an exact projection but we may compute a point p on the convex hull of the columns of A that minimizes the worst-case distance to b. This approach is known as robust optimization. Formally, we solve the min-max problem

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & \max_{\theta \in \Theta} \|Ax - (b + \theta)\| \\ \text{subject to} & \sum_{i=1}^{n} x_i = 1, \\ & x \succeq 0, \end{array}$$

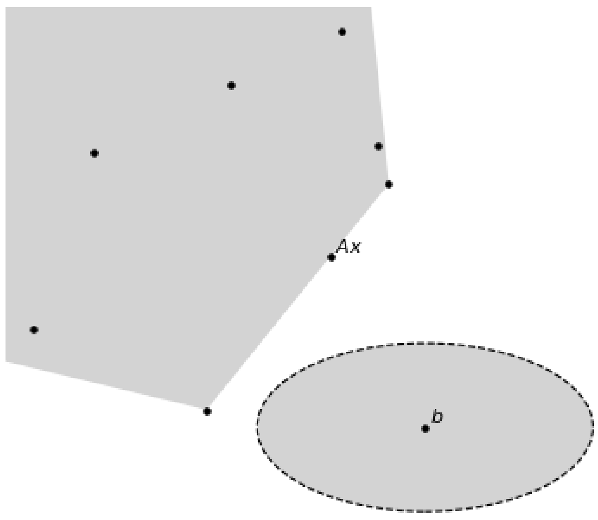
where $\Theta = \{\theta \mid L\theta \leq 1\}$ is an ellipsoidal *perturbation set* (for some invertible matrix L):

```
#!/usr/bin/env python3

import numpy as np
import picos as pc
from matplotlib import pyplot
from matplotlib.patches import Ellipse
from scipy import spatial

# Make the result reproducible.
np.random.seed(12)

# Define the data.
n = 20
A = np.random.rand(2, n)
b = np.array([1, 0])

# Define an ellipsoidal uncertainty set Θ and a perturbation parameter θ.
# The perturbation is later added to the data, rendering it uncertain.
Theta = pc.uncertain.ConicPerturbationSet("θ", 2)
Theta.bound(  # Let ‖Lθ‖ ≤  1.
  abs([[ 5,  0],
       [ 0, 10]] * Theta.element) <= 1
)
theta = Theta.compile()

# Define the decision variable.
x = pc.RealVariable("x", n)

# Define and solve the problem.
P = pc.Problem()
P.minimize = abs(A*x - (b + theta))
P += pc.sum(x) == 1, x >= 0
P.solve(solver="cvxopt")

# Obtain the projection point.
p = (A*x).np

# Plot the results.
V = spatial.ConvexHull(A.T).vertices
E = Ellipse(b, 0.4, 0.2, color="lightgray", ec="k", ls="--")
figure = pyplot.figure(figsize=(8.7, 4))
axis = figure.gca()
axis.add_artist(E)
```

(continues on next page)

(continued from previous page)

```
axis.set_aspect("equal")
axis.set_xlim(0.5, 1.21)
axis.set_ylim(-0.11, 0.5)
pyplot.axis("off")
pyplot.fill(A.T[V, 0], A.T[V, 1], "lightgray")
pyplot.plot(A.T[:, 0], A.T[:, 1], "k.")
pyplot.plot(*zip(b, p), "k.")
pyplot.annotate("$\mathrm{conv} \{a_1, \ldots, a_n\}$", [0.25, 0.5])
pyplot.annotate("$b$", b + 1/200)
pyplot.annotate("$Ax$", p + 1/200)
pyplot.tight_layout()
pyplot.show()
```



Example notes

- One can also scale and shift the parameter obtained from a `UnitBallPerturbationSet` to obtain ellipsoidal uncertainty. Its parent class `ConicPerturbationSet` that we showcased is more versatile and can represent any conically bounded perturbation set through repeated use of its `bound` method.
- A report of the robust and distributionally robust optimization models supported by PICOS and their mathematical background is found in *[1]*.

3.1.3 Optimal Minecraft mob farm

Minecraft is a popular sandbox video game in which some players aim to build efficient automated factories, referred to as *farms*. One type of farm waits for hostile creatures (*mobs*) to appear on a platform, then pushes them off the platform with a water dispenser in the center to collect any valuables that they might carry. Such a farm is threatened by the possibility of Spiders to appear, which are too large for the collection mechanism to handle. Fortunately, the Spider requires a 3×3 area to spawn on while the other mobs require just a single free 1×1 cell, so Spider spawns can be prevented by blocking off some of the platform's cells.

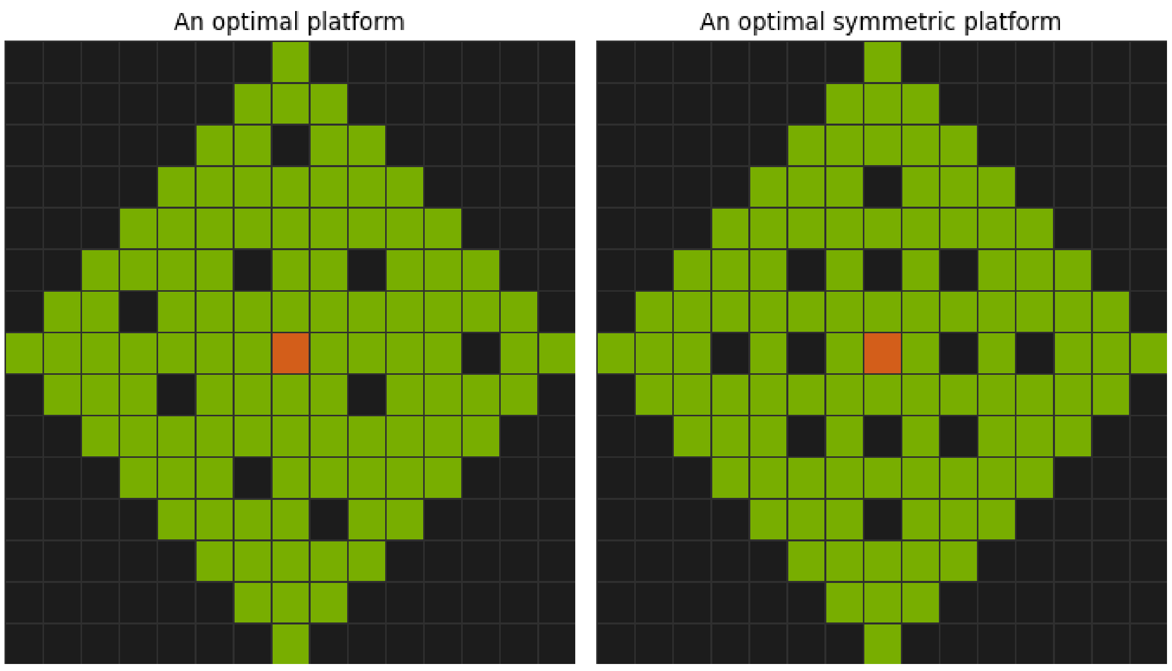
In the following we compute an optimal platform that maximizes the number of cells that mobs can spawn on while admitting no 3×3 spawnable region for Spiders. We further compute an optimal highly symmetric (w.r.t. both axes and diagonals) solution for those who value looks over efficiency:

```
#!/usr/bin/env python3

import picos as pc
from matplotlib import colors, pyplot

# Represent the spawning platform by a 15×15 binary matrix variable S where a
# one represents a spawnable field and a zero one that is not spawnable.
S = pc.BinaryVariable("S", (15, 15))

# Maximize the number of spawnable blocks.
P = pc.Problem("Optimal Mob Farm")
P.maximize = pc.sum(S)

# The actual platform is shaped like a diamond of cells with taxicab distance
# of at most seven from the center block. Mark all other cells not spawnable.
P += [
    S[x, y] == 0
    for x in range(S.shape[0])
    for y in range(S.shape[1])
    if abs(x - 7) + abs(y - 7) > 7
]

# The center block is not spawnable due to the water dispenser.
P += S[7, 7] == 0

# Additionally, we require that there is no 3x3 spawnable area.
P += [
    sum([
        S[a, b]
        for a in range(x - 1, x + 2)
        for b in range(y - 1, y + 2)
    ]) <= 8
    for x in range(1, S.shape[0] - 1)
    for y in range(1, S.shape[1] - 1)
]

# Solve the problem and store the optimal platform.
P.solve(solver="glpk")
S_opt = S.np

# Now modify the problem to require a more symmetric solution.
P += [S[x, :] == S[14 - x, :] for x in range(S.shape[0] // 2)]  # Vertical.
P += S == S.T  # Diagonal.

# Re-solve the updated problem.
P.solve()
S_sym = S.np

# Display both solutions.
figure, axes = pyplot.subplots(ncols=2, figsize=(8.7, 5))
titles = ["An optimal platform", "An optimal symmetric platform"]
cmap = colors.ListedColormap(["#1c1c1c", "#78ae00", "#d35e1a"])

for axis, title, solution in zip(axes, titles, [S_opt, S_sym]):
  solution[7, 7] = 2  # Mark the center.
  axis.axis("off")
  axis.set_title(title)
```

(continues on next page)

(continued from previous page)

```
  axis.pcolormesh(solution, edgecolor="#2f2f2f", linewidth=0.5, cmap=cmap)

pyplot.tight_layout()
pyplot.show()
```



Example notes

- Excluding the center, the platform has 112 cells. The solutions show that an optimal platform has 9 obstacles and 103 free cells (92.0%) while an optimal symmetric platform has 12 obstacles and thus only 100 free cells (89.3%).
- The two symmetry conditions require symmetry along one axis and one main diagonal, respectively. Symmetry along the remaining axis and diagonal is obtained implicitly. With an adjustment it can be seen that only requiring axial symmetry does not increase efficiency.

3.1.4 References

1. “Robust conic optimization in Python”, M. Stahlberg, Master’s thesis, 2020.

3.2 Graph flow and cut problems

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 required. 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 pc
import networkx as nx
import pylab
import random
```

(continues on next page)

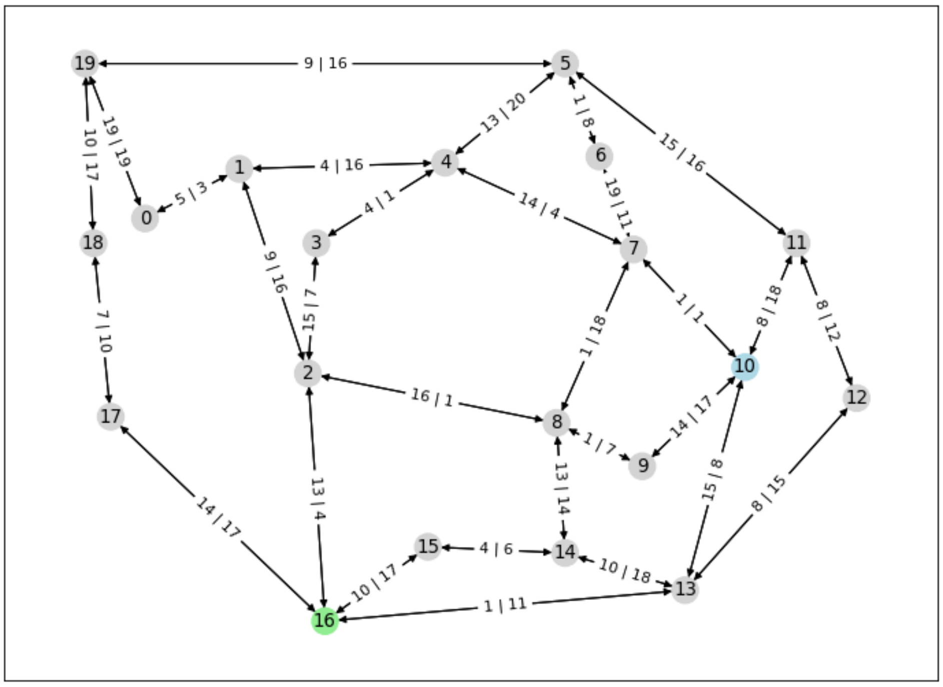
(continued from previous page)

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

# 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=pc.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]}
```

(continues on next page)

(continued from previous page)

```
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.

3.2.1 Max-flow (LP)

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, $\forall n \in V \setminus \{s, t\},\ \sum_{(i,n)\in E} f((i, n)) = \sum_{(n,j)\in E} f((n, j))$.

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

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

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

```
maxflow=pc.Problem()

# Add the flow variables.
f={}
```

(continues on next page)

(continued from previous page)

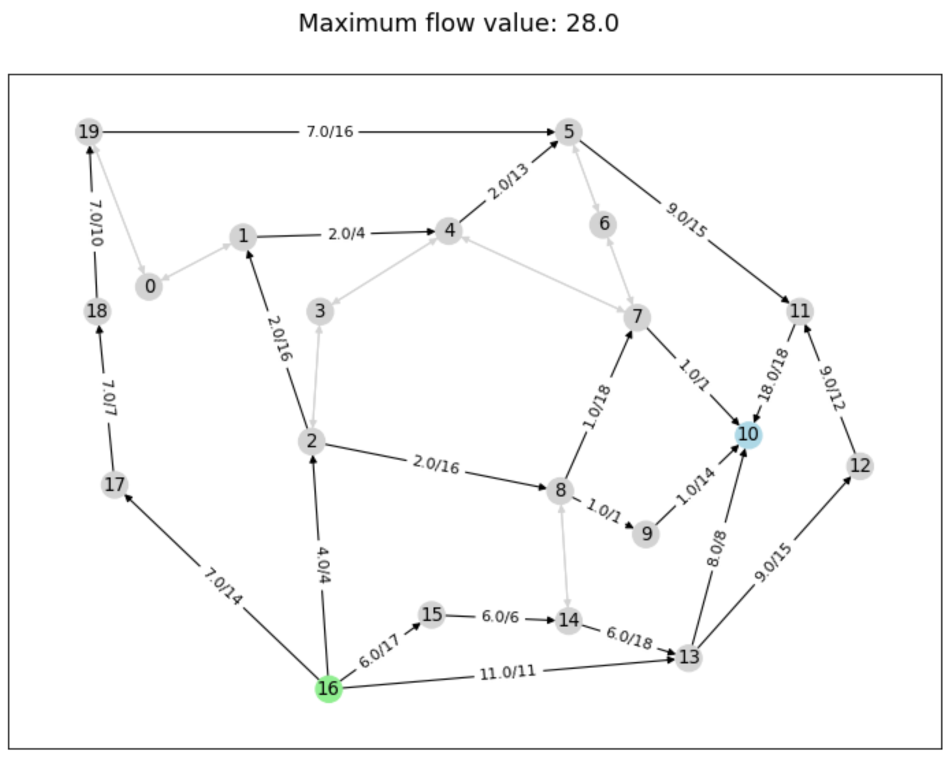
```
for e in G.edges():
  f[e]=maxflow.add_variable('f[{0}]'.format(e))

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

# Enforce edge capacities.
maxflow.add_list_of_constraints([f[e] <= cc[e] for e in G.edges()])

# Enforce flow conservation.
maxflow.add_list_of_constraints([
    pc.sum([f[p,i] for p in G.predecessors(i)])
    == pc.sum([f[i,j] for j in G.successors(i)])
    for i in G.nodes() if i not in (s,t)])

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

# Set sink flow at t.
maxflow.add_constraint(
  pc.sum([f[p,t] for p in G.predecessors(t)])
  == pc.sum([f[t,j] for j in G.successors(t)]) + F)

# Enforce flow nonnegativity.
maxflow.add_list_of_constraints([f[e] >= 0 for e in G.edges()])

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

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

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

```
maxflow2=pc.Problem()

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

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

# Enforce all flow constraints at once.
maxflow2.add_constraint(pc.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(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.

3.2.2 Min-cut (LP)

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

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

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

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

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

```
mincut=pc.Problem()

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

# 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()])

# 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()])

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

mincut.solve(solver='glpk')

# Determine the cut edges and node sets.
# Rounding is done because solvers might return near-optimal solutions due to
```

(continues on next page)

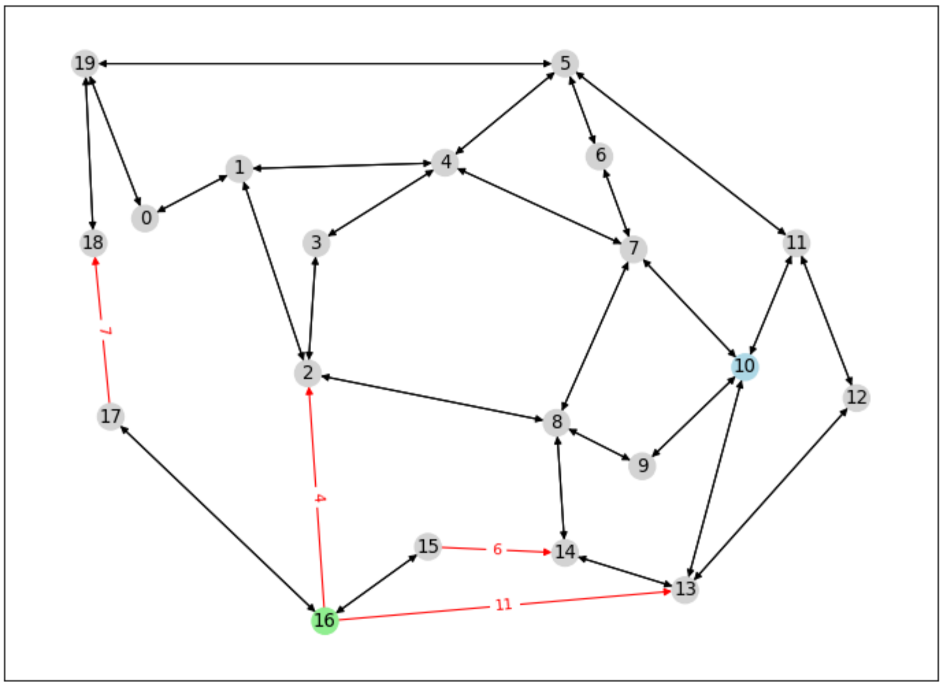
(continued from previous page)

```
# 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]
```

Let us now draw the minimum 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()
```

Minimum cut value: 28.0
S: [15, 16, 17], T: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19]



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

```
# capacited flow constraint
capaflow = maxflow.get_constraint((0,))
dualcut = [
  e for i, e in enumerate(G.edges()) if abs(capaflow[i].dual - 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 - 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) < 1e-6]
```

Let's see how this dual-derived cut looks like:

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

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

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

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

# Show the dual cut value and the partition.
fig.suptitle("Minimum cut value: {}\nS: {}, T: {}".format(
  sum(cc[e] for e in dualcut), Sdual, Tdual), 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.

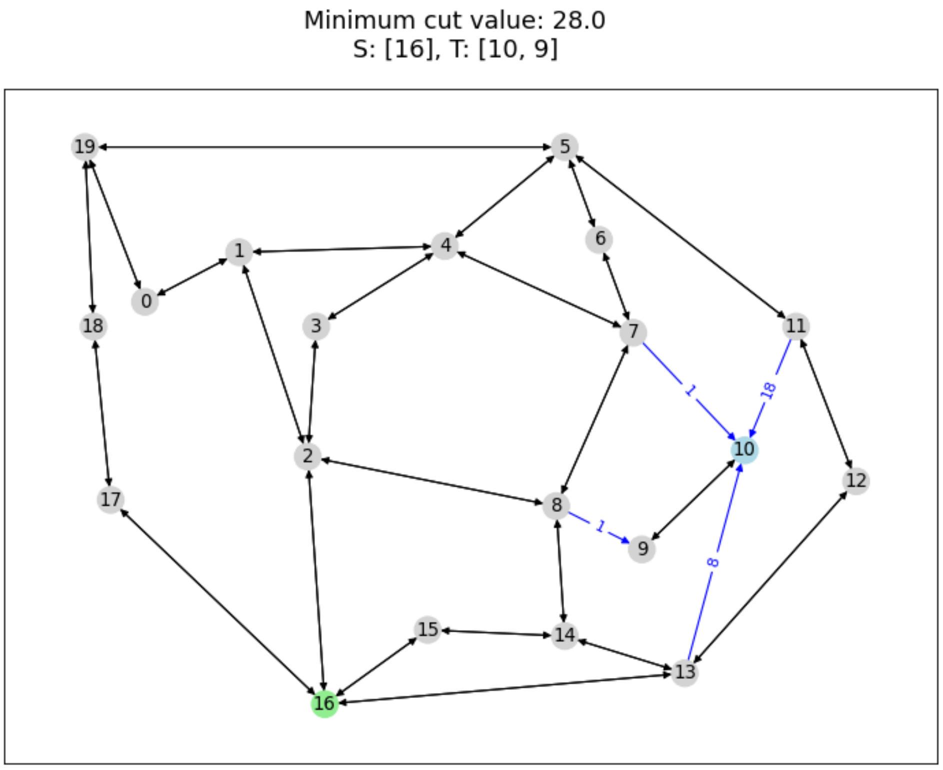
3.2.3 Multicut (MIP)

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 $(s,t) \in \mathcal{P} = \{(s_1,t_1),\dots,(s_k,t_k))\}$, 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.



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

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=pc.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), vtype='binary')
```

(continues on next page)

(continued from previous page)

```

# 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()])

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

# Set the sink potentials to zero.
multicut.add_list_of_constraints([p[s][t] == 0 for (s,t) in pairs])

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

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

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

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

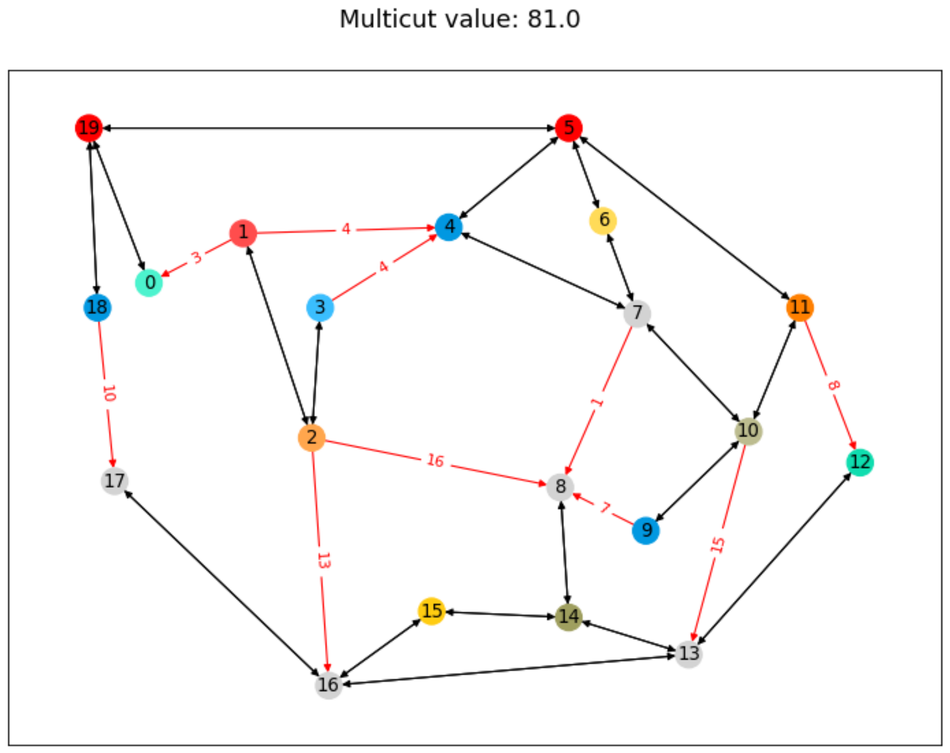
Let us now draw the multicut:

```
# 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')
```

(continues on next page)

(continued from previous page)

```
# 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.

3.2.4 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 $\frac{1}{4}x^T Lx$, 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:

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

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 = pc.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=pc.new_param('L', LL)

# Constrain X to have ones on the diagonal.
maxcut.add_constraint(pc.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(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
```

(continues on next page)

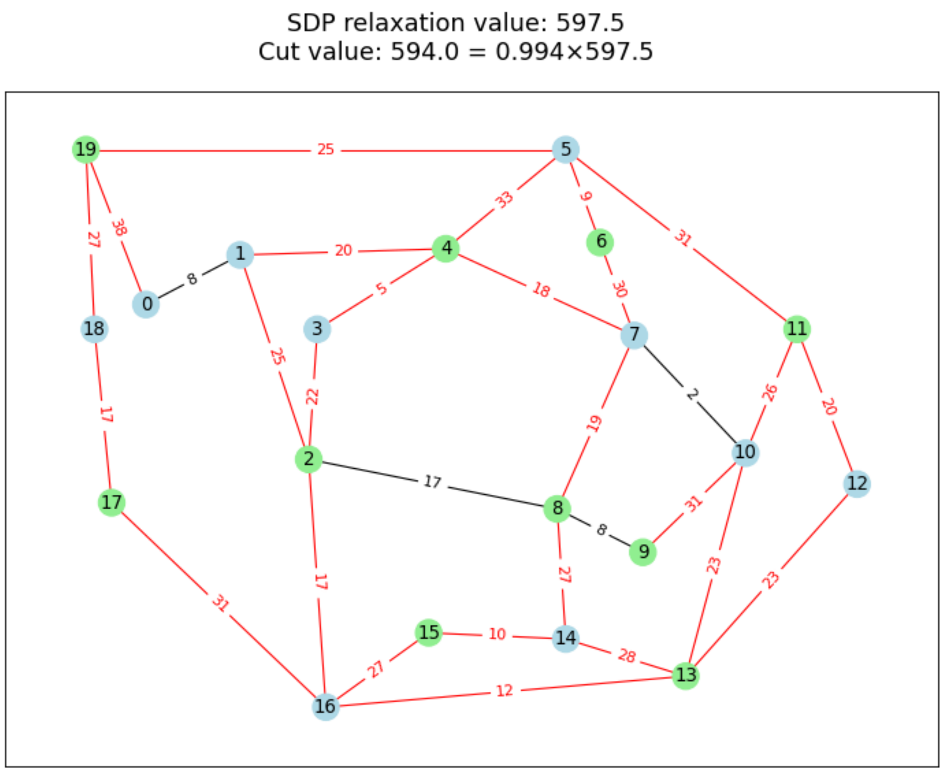
(continued from previous page)

```
# 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
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.



3.2.5 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.

3.3 Complex Semidefinite Programming

PICOS supports complex semidefinite programming as of version 1.0.1. It was overhauled in version 2.0 to provide some of the features showcased below. This extension of semidefinite programming to the complex domain was introduced by Goemans and Williamson *[1]* in order to pose relaxations of combinatorial optimization problems. Applications include quantum information theory *[2]* and the phase recovery problem in signal processing *[3]*.

Complex problems can be defined in PICOS using the complex-valued variable types `ComplexVariable` and `HermitianVariable`:

```
>>> from picos import ComplexVariable, HermitianVariable
>>> z = ComplexVariable("z", 4)
>>> H = HermitianVariable("H", 4)
>>> z
<4×1 Complex Variable: z>
>>> H
<4×4 Hermitian Variable: H>
>>> z.real
```

(continues on next page)

(continued from previous page)

```
<4×1 Real Linear Expression: Re(z)>
>>> z.imag
<4×1 Real Linear Expression: Im(z)>
```

Their value can be set and retrieved as in the real case but may contain an imaginary part:

```
>>> z.value = [1, 2+2j, 3+3j, 4j]
>>> z.value  # Note the CVXOPT typecode of 'z'.
<4x1 matrix, tc='z'>
>>> print(z)
[ 1.00e+00-j0.00e+00]
[ 2.00e+00+j2.00e+00]
[ 3.00e+00+j3.00e+00]
[ 0.00e+00+j4.00e+00]
>>> z.real.value
<4x1 matrix, tc='d'>
>>> print(z.real)
[ 1.00e+00]
[ 2.00e+00]
[ 3.00e+00]
[ 0.00e+00]
>>> print(z.imag)
[ 0.00e+00]
[ 2.00e+00]
[ 3.00e+00]
[ 4.00e+00]
```

Just like real variables are the simplest form of an *AffineExpression*, complex variables are represented to you as instances of *ComplexAffineExpression*. Most notably this gives access to complex conjugation and hermitian transposition:

```
>>> z.conj
<4×1 Complex Linear Expression: conj(z)>
>>> z.H
<1×4 Complex Linear Expression: zᴴ>
```

Internally complex variables are represented as real variable vectors:

```
>>> z.dim  # Twice its dimension on the complex field.
8
>>> H.dim  # The same dimension as an arbitrary real matrix of same shape.
16
```

Note that in the hermitian case, we get away with just $4 \cdot 4 = 16$ *real* scalar variables due to the vectorization used. This leads to a smaller footprint when the problem is passed to a solver.

Unlike real-valued variables, *ComplexVariable* and *HermitianVariable* do not accept variable bounds at creation, and any properly complex expression formed from them cannot appear on either side of an affine inequality constraint or as an objective function. However, PICOS detects when you supply a real-valued expression in any of these places even if it was created from complex expressions:

```
>>> A = ~z*~z.H  # Use the current value of z to create a constant 4×4 matrix.
>>> A
<4×4 Complex Constant: [z]·[zᴴ]>
>>> A.hermitian  # By construction this matrix is hermitian.
True
>>> (H|A)  # Create a complex expression involving H.
```

(continues on next page)

(continued from previous page)

```
<1×1 Complex Linear Expression: ⟨H, [z]·[zᴴ]⟩>
>>> (H|A).isreal  # On closer inspection, it is always real-valued.
True
>>> (H|A).refined  # This means it can be "refined" to a real expression.
<1×1 Real Linear Expression: ⟨H, [z]·[zᴴ]⟩>
>>> (H|A) >= 0  # Refinement happens automatically wherever necessary.
<1×1 Affine Constraint: ⟨H, [z]·[zᴴ]⟩ ≥ 0>
>>> H == A  # Equalities involving complex expressions can be posed as normal.
<4×4 Complex Equality Constraint: H = [z]·[zᴴ]>
```

Complex linear matrix inequalities are created just as in the real case with the overloaded << and >> operators representing the Loewner order:

```
>>> H >> 0
<4×4 Complex LMI Constraint: H ≽ 0>
```

Since solvers at this time generally do not support complex optimization, PICOS transforms such a constraint to an equivalent real LMI during solution search. Only to demonstrate this behavior, we do it manually:

```
>>> from picos import Options
>>> from picos.constraints import ComplexLMIConstraint
>>> P = ComplexLMIConstraint.RealConversion.convert(H >> 0, Options())
>>> P.get_constraint(0)
<8×8 LMI Constraint: [Re(H), -Im(H); Im(H), Re(H)] ≽ 0>
```

3.3.1 Fidelity in Quantum Information Theory

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

The fidelity between two (hermitian) positive semidefinite operators P and Q is defined as

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

where the trace norm $\|\cdot\|_{\text{tr}}$ 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:

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

This model can be implemented in PICOS as follows:

```
import numpy
import picos

# Create a positive semidefinite constant P.
_P = picos.Constant([
    [ 1  -1j,  2  +2j,  1     ],
    [     3j,     -2j, -1  -1j],
    [ 1  +2j, -0.5+1j,  1.5   ]])
P = (_P*_P.H).renamed("P")

# Create a positive semidefinite constant Q.
_Q = picos.Constant([
    [-1  -2j,      2j,  1.5   ],
```

(continues on next page)

(continued from previous page)

```
    [ 1  +2j,     -2j,  2.0-3j],
    [ 1  +2j, -1  +1j,  1  +4j]])
Q = (_Q*_Q.H).renamed("Q")

# Define the problem.
F = picos.Problem()
Z = picos.ComplexVariable("Z", P.shape)
F.set_objective("max", 0.5*picos.trace(Z + Z.H))
F.add_constraint(((P & Z) // (Z.H & Q)) >> 0)

print(F)

# Solve the problem.
F.solve(solver = "cvxopt")

print("\nOptimal value:", round(F, 4))
print("Optimal Z:", Z.value, sep="\n")

# Also compute the fidelity via NumPy for comparison.
PP  = numpy.matrix(P.value)
QQ  = numpy.matrix(Q.value)
S,U = numpy.linalg.eig(PP)
sqP = U * numpy.diag([s**0.5 for s in S]) * U.H  # Square root of P.
S,U = numpy.linalg.eig(QQ)
sqQ = U * numpy.diag([s**0.5 for s in S]) * U.H  # Square root of Q.
Fnp = sum(numpy.linalg.svd(sqP * sqQ)[1])  # Trace-norm of sqrt(P)·sqrt(Q).

print("Fidelity F(P,Q) computed by NumPy:", round(Fnp, 4))
```

```
Complex Semidefinite Program
  maximize 0.5·tr(Z + Zᴴ)
  over
    3×3 complex variable Z
  subject to
    [P, Z; Zᴴ, Q] ≽ 0

Optimal value: 39.8938
Optimal Z:
[ 1.06e+01+j2.04e+00 -7.21e+00+j5.77e+00  3.58e+00-j8.10e+00]
[-8.26e+00-j2.13e+00  1.65e+01+j3.61e-01  8.59e-02-j2.29e+00]
[-1.38e+00+j6.42e+00 -5.65e-01+j1.55e+00  1.28e+01-j2.40e+00]

Fidelity F(P,Q) computed by NumPy: 39.8938
```

3.3.2 Phase Recovery in Signal Processing

This section is inspired by *[3]*.

The goal of the phase recovery problem is to reconstruct the complex phase of a vector given only the magnitudes of some linear measurements. This problem can be formulated as a non-convex optimization problem, and the authors of *[3]* have proposed a complex semidefinite relaxation similar to the well known relaxation of the **Max-Cut Problem**: 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:

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

Note that U must be hermitian ($U \in \mathbb{H}_n$). We obtain an exact solution u 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:

```python
import cvxopt
import numpy
import picos

# Make the output reproducible.
cvxopt.setseed(1)

# Generate an arbitrary rank-deficient hermitian matrix M.
n, rank = 5, 4
m = cvxopt.normal(n, rank) + 1j*cvxopt.normal(n, rank)
M = picos.Constant("M", m*m.H)

# Define the problem.
P = picos.Problem()
U = picos.HermitianVariable("U", n)
P.set_objective("min", (U | M))
P.add_constraint(picos.maindiag(U) == 1)
P.add_constraint(U >> 0)

print(P)

# Solve the problem.
P.solve(solver="cvxopt")

print("\nOptimal U:", U, sep="\n")

# Determine the rank of U.
S, V = numpy.linalg.eig(U.value)
Urnk = len([s for s in S if abs(s) > 1e-6])

print("\nrank(U) =", Urnk)
```

```
Complex Semidefinite Program
  minimize ⟨U, M⟩
  over
    5×5 hermitian variable U
  subject to
    maindiag(U) = [1]
    U ⪰ 0

Optimal U:
[ 1.00e+00-j0.00e+00  6.31e-01-j7.76e-01 -8.84e-01+j4.68e-01  6.23e-01-j7.82e-01  7.
↪52e-01+j6.59e-01]
[ 6.31e-01+j7.76e-01  1.00e+00-j0.00e+00 -9.20e-01-j3.91e-01  1.00e+00-j9.69e-03 -3.
↪75e-02+j9.99e-01]
[-8.84e-01-j4.68e-01 -9.20e-01+j3.91e-01  1.00e+00-j0.00e+00 -9.17e-01+j4.00e-01 -3.
↪56e-01-j9.34e-01]
```

(continues on next page)

(continued from previous page)

```
[ 6.23e-01+j7.82e-01  1.00e+00+j9.69e-03 -9.17e-01-j4.00e-01  1.00e+00-j0.00e+00 -4.
↪72e-02+j9.99e-01]
[ 7.52e-01-j6.59e-01 -3.75e-02-j9.99e-01 -3.56e-01+j9.34e-01 -4.72e-02-j9.99e-01  1.
↪00e+00-j0.00e+00]

rank(U) = 1
```

3.3.3 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, https://cs.uwaterloo.ca/~watrous/LectureNotes/CS766.Fall2011/08.pdf.

3.4 Quantum Relative Entropy Programming

PICOS supports quantum relative entropy programming as of version 2.5.0 when used with the solver QICS. These are convex optimization problems which minimizing over the quantum (Umegaki) relative entropy, which is defined as

$$S(X\|Y) = \text{Tr}(X\log(X) - X\log(Y)),$$

over positive semidefinite matrices $X \in \mathbb{H}^n_+$ and $Y \in \mathbb{H}^n_+$. This function is jointly convex in both of its arguments. In PICOS, this function is represented by the expression `quantrelentr`.

Below, we show two examples of quantum relative entropy programs which arise in quantum information theory. These are taken from the QICS documentation, which contain many other examples of quantum relative entropy programs which can be solved using PICOS.

3.4.1 Relative entropy of entanglement

Consider a bipartite quantum state $X \in \mathbb{H}^{n_1 n_2}$. The relative entropy of entanglement aims to quantify how entangled X is by measuring the distance, in the quantum relative entropy sense, to the set of separable states.

However, describing the set of separable states is NP-hard in general. Therefore, it is useful to use a relaxation of this condition known as the positive partial transpose (PPT) criterion *[1]*

$$\mathsf{PPT} = \{X \in \mathbb{H}^{n_1 n_2} : X^{T_2} \succeq 0\},$$

where $X \mapsto X^{T_2}$ denotes the partial transpose with respect to the second subsystem. The (approximate) relative entropy of entanglement is then given as the optimal value of

$$\begin{array}{ll} \underset{Y \in \mathbb{H}^{n_1 n_2}}{\text{minimize}} & S(X\|Y) \\ \text{subject to} & \text{Tr}(Y) = 1 \\ & Y^{T_2} \succeq 0 \\ & Y \succeq 0. \end{array}$$

This can be implemented in PICOS as follows:

```python
import picos

# Create a quantum state X.
X = picos.Constant("X", [
    [0.5, 0.0, 0.0, 0.5],
    [0.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0],
    [0.5, 0.0, 0.0, 0.5]])

# Define the problem.
P = picos.Problem()
Y = picos.SymmetricVariable("Y", 4)

P.set_objective("min", picos.quantrelentr(X, Y))
P.add_constraint(picos.trace(Y) == 1.0)
P.add_constraint(Y.partial_transpose(1) >> 0)

print(P)

# Solve the problem.
P.solve(solver="qics")

print("\nRelative entropy of entanglement of X:", round(P, 4))
```

```
Quantum Relative Entropy Program
  minimize S(X‖Y)
  over
    4×4 symmetric variable Y
  subject to
    tr(Y) = 1
    Y.{[2×2]⊗[2×2]ᵀ} ⪰ 0

Relative entropy of entanglement of X: 0.6931
```

3.4.2 Entanglement-assisted channel capacity

When using a quantum channel to transmit information, we are often interested in the maximum rate of information we can transmit in a way that is robust of noise. Depending on what quantum resources are used, there are different theorems which describe this limit.

For a quantum channel described by a Stinespring represntation $\mathcal{N}(X) = \mathrm{Tr}_2(VXV^\dagger)$, the entanglement- assisted channel capacity *[2]* is given by the optimal value of

$$\begin{array}{ll} \underset{X\in\mathbb{H}^n}{\text{maximize}} & S(VXV^\dagger) - S(\mathrm{Tr}_1(VXV^\dagger)) + S(\mathrm{Tr}_2(VXV^\dagger)) \\ \text{subject to} & \mathrm{Tr}(X) = 1 \\ & X \succeq 0. \end{array}$$

The objective function is known as the quantum mutual information, and can be modelled in PICOS using the `quantcondentr` and `quantentr` expressions.

```python
import math
import picos

# Define Stinespring isometry for amplitude damping channel.
gamma = 0.5
```

(continues on next page)

(continued from previous page)

```
V = picos.Constant("V", [
    [1., 0.                ],
    [0., math.sqrt(1-gamma)],
    [0., math.sqrt(gamma)  ],
    [0., 0.                ]
])

# Define the problem.
P = picos.Problem()
X = picos.SymmetricVariable("X", 2)

obj1 = picos.quantcondentr(V*X*V.T, 1)
obj2 = picos.quantentr(picos.partial_trace(V*X*V.T, 0))

P.set_objective("max", obj1 + obj2)
P.add_constraint(picos.trace(X) == 1)
P.add_constraint(X >> 0)

print(P)

# Solve the problem.
P.solve(solver="qics")

print("\nEntanglement-assisted channel capacity:", round(P, 4))
```

```
Quantum Relative Entropy Program
  maximize S(V·X·Vᵀ) - S((V·X·Vᵀ).{[2×2]⊗tr([2×2])}) + S((V·X·Vᵀ).{tr([2×2])⊗[2×2]})
  over
    2×2 symmetric variable X
  subject to
    tr(X) = 1
    X ≽ 0

Entanglement-assisted channel capacity: 0.6931
```

3.4.3 Quantum key distribution

When designing a quantum cryptographic protocol, we are interested in computing the quantum key rate of a given protocol which allows us to certify the security of the protocol. This quantum key rate can be computed by solving the quantum relative entropy program *[3]*

$$\begin{array}{ll} \underset{X\in\mathbb{H}^n}{\text{minimize}} & S(\mathcal{G}(X)\|\mathcal{Z}(\mathcal{G}(X))) \\ \text{subject to} & \operatorname{Tr}(A_iX) = b_i, \quad i = 1,\ldots,p \\ & X \succeq 0. \end{array}$$

where $\mathcal{G}$ is a completely positive linear map, $\mathcal{Z}$ is the pinching map which maps off-diagonal blocks of a , block-matrix to zero, and A_i and b_i encode a set of experimental constraints.

In PICOS, this slice of the quantum relative entropy function can be modelled using the *quantkeydist* expression. Below, we show how the key rate of the entanglement assisted BB84 protocol from *[4]* can be computed using PICOS.

```
import numpy
import picos
```

(continues on next page)

(continued from previous page)

```
# Define entanglement assisted BB84 protocol.
qx = 0.25
qz = 0.75

X0 = numpy.array([[.5,  .5], [ .5, .5]])
X1 = numpy.array([[.5, -.5], [-.5, .5]])
Z0 = numpy.array([[1.,  0.], [ 0., 0.]])
Z1 = numpy.array([[0.,  0.], [ 0., 1.]])

Ax = numpy.kron(X0, X1) + numpy.kron(X1, X0)
Az = numpy.kron(Z0, Z1) + numpy.kron(Z1, Z0)

# Define the problem.
P = picos.Problem()
X = picos.SymmetricVariable("X", 4)

P.set_objective("min", picos.quantkeydist(X))
P.add_constraint(picos.trace(X) == 1)
P.add_constraint((X | Ax) == qx)
P.add_constraint((X | Az) == qz)

print(P)

# Solve the problem.
P.solve(solver="qics")

print("\nebBB84 key rate:", round(P, 4))
```

```
Quantum Relative Entropy Program
  minimize S(X‖Z(X))
  over
    4×4 symmetric variable X
  subject to
    tr(X) = 1
    ⟨X, [4×4]⟩ = 0.25
    ⟨X, [4×4]⟩ = 0.75

ebBB84 key rate: 0.1308
```

3.4.4 References

1. "Separability of mixed states: necessary and sufficient conditions," M. Horodecki, P. Horodecki, and R. Horodecki, Physics Letters A, vol. 223, no. 1, pp. 1–8, 1996.
2. "Entanglement-assisted capacity of a quantum channel and the reverse Shannon theorem," C. H. Bennett, P. W. Shor, J. A. Smolin, and A. V. Thapliyal, IEEE transactions on Information Theory, vol. 48, no. 10, pp. 2637–2655, 2002.
3. "Reliable numerical key rates for quantum key distribution", A. Winick, N. Lutkenhaus, and P. J. Coles. Quantum, vol. 2, p. 77, 2018.
4. "Quantum key distribution rates from non-symmetric conic optimization", L. A. Gonzalez, et al. arXiv preprint arXiv:2407.00152, 2024.

3.5 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
>>> #---------------------------------#
>>> # First generate some data :      #
>>> #       _ a list of 8 matrices A  #
>>> #       _ a vector c              #
>>> #---------------------------------#
>>> A = [cvx.matrix([[1,0,0,0,0],
...                  [0,3,0,0,0],
...                  [0,0,1,0,0]]),
...      cvx.matrix([[0,0,2,0,0],
...                  [0,1,0,0,0],
...                  [0,0,0,1,0]]),
...      cvx.matrix([[0,0,0,2,0],
...                  [4,0,0,0,0],
...                  [0,0,1,0,0]]),
...      cvx.matrix([[1,0,0,0,0],
...                  [0,0,2,0,0],
...                  [0,0,0,0,4]]),
...      cvx.matrix([[1,0,2,0,0],
...                  [0,3,0,1,2],
...                  [0,0,1,2,0]]),
...      cvx.matrix([[0,1,1,1,0],
...                  [0,3,0,1,0],
...                  [0,0,2,2,0]]),
...      cvx.matrix([[1,2,0,0,0],
...                  [0,3,3,0,5],
...                  [1,0,0,2,0]]),
...      cvx.matrix([[1,0,3,0,1],
...                  [0,3,2,0,0],
...                  [1,0,0,2,0]])
... ]
>>> c = cvx.matrix([1,2,3,4,5])
```

3.5.1 Multi-response c-optimal design (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:

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

```
>>> # create the problem, variables and params
>>> c_primal_SOCP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> cc = picos.Constant('c', c)
>>> z  = [picos.RealVariable('z[{0}]'.format(i), AA[i].size[1])
...       for i in range(s)]
>>> mu = picos.RealVariable('mu', s)
```

```
>>> # define the constraints and objective function
>>> cones = c_primal_SOCP.add_list_of_constraints([abs(z[i]) <= mu[i] for i in␣
↪range(s)])
>>> lin   = c_primal_SOCP.add_constraint(picos.sum([AA[i] * z[i] for i in range(s)])␣
↪== cc)
>>> c_primal_SOCP.set_objective('min', mu.sum)
>>> print(c_primal_SOCP)
Second Order Cone Program
  minimize ∑(mu)
  over
    3×1 real variable z[i] ∀ i ∈ [0...7]
    8×1 real variable mu
  subject to
    ‖z[i]‖ ≤ mu[i] ∀ i ∈ [0...7]
    ∑(A[i]·z[i] : i ∈ [0...7]) = c
```

```
>>> #solve the problem and retrieve the optimal weights of the optimal design.
>>> solution = c_primal_SOCP.solve(solver='cvxopt')
>>> mu = mu.value
>>> w = mu / sum(mu) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[...]
```

(continues on next page)

(continued from previous page)

```
[...]
[ 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 tutorial:

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

```
>>> # create the problem, variables and params
>>> c_dual_SOCP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> cc = picos.Constant('c',c)
>>> u  = picos.RealVariable('u',c.size)
>>> # define the constraints and objective function
>>> cones = c_dual_SOCP.add_list_of_constraints(
...           [abs(AA[i].T*u)<=1 for i in range(s)])
>>> c_dual_SOCP.set_objective('max', (cc | u))
>>> print(c_dual_SOCP)#
Second Order Cone Program
  maximize ⟨c, u⟩
  over
    5×1 real variable u
  subject to
    ‖A[i]ᵀ·u‖ ≤  1 ∀ i ∈  [0...7]
>>> #solve the problem and retrieve the weights of the optimal design
>>> solution = c_dual_SOCP.solve(solver='cvxopt')
>>> mu = [cons.dual[0] for cons in cones] #Lagrangian duals of the SOC constraints
>>> mu = cvx.matrix(mu)
>>> w=mu/sum(mu) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[...]
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]
```

3.5.2 Single-response c-optimal design (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
>>> c_primal_LP = picos.Problem()
>>> A1 = [cvx.sparse(a[:,i],tc='d') for i in range(3) for a in A[4:]] #12 column␣
↪vectors
>>> A1 = A1[:-1] # remove the last design point (it is the same as the last-but-one)
>>> s = len(A1)
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...        for i, Ai in enumerate(A1)] # each AA[i].T is a 1 x 5 observation matrix
>>> cc = picos.Constant('c', c)
>>> z = [picos.RealVariable('z[{0}]'.format(i), 1) for i in range(s)]
>>> mu = picos.RealVariable('mu', s)
```

```
>>> #define the constraints and objective function
>>> abs_con = c_primal_LP.add_list_of_constraints([abs(z[i]) <= mu[i] for i in␣
↪range(s)])
>>> lin_con = c_primal_LP.add_constraint(picos.sum([AA[i]*z[i] for i in range(s)]) ==␣
↪cc)
>>> c_primal_LP.set_objective('min', mu.sum)
```

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:

```
>>> print(c_primal_LP)
Linear Program
  minimize ∑(mu)
  over
    1×1 real variable z[i] ∀ i ∈ [0...10]
    11×1 real variable mu
  subject to
    |z[i]| ≤ mu[i] ∀ i ∈ [0...10]
    ∑(A[i]·z[i] : i ∈ [0...10]) = c
```

```
>>> #solve the problem and retrieve the weights of the optimal design
>>> solution = c_primal_LP.solve(solver='cvxopt')
>>> mu = mu.value
>>> w = mu / sum(mu) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[...]
[...]
[ 3.37e-02]
[...]
[ 2.79e-01]
[ 1.18e-01]
```

(continues on next page)

(continued from previous page)

```
[ 2.76e-01]
[...]
[ 2.93e-01]
```

3.5.3 SDP formulation of c-optimal design

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:

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

```
>>> # create the problem, variables and params
>>> c_primal_SDP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> cc = picos.Constant('c', c)
>>> mu = picos.RealVariable('mu',s)
>>> # define the constraints and objective function
>>> lmi = c_primal_SDP.add_constraint(
...         picos.sum([mu[i] * AA[i] * AA[i].T for i in range(s)]) >> cc*cc.T)
>>> lin_cons = c_primal_SDP.add_constraint(mu >= 0)
>>> c_primal_SDP.set_objective('min', mu.sum)
>>> print(c_primal_SDP)
Semidefinite Program
  minimize ∑(mu)
  over
    8×1 real variable mu
  subject to
    ∑(mu[i]·A[i]·A[i]ᵀ : i ∈ [0...7]) ≽ c·cᵀ
    mu ≥ 0
```

```
>>> #solve the problem and retrieve the weights of the optimal design
>>> solution = c_primal_SDP.solve(solver='cvxopt')
>>> w = mu.value
>>> w = w / sum(w) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[...]
```

(continues on next page)

(continued from previous page)

```
[...]
[ 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 tutorial:

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

```
>>> #create the problem, variables and params
>>> c_dual_SDP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> cc = picos.Constant('c', c)
>>> m  = c.size[0]
>>> X  = picos.SymmetricVariable('X',(m,m))
```

```
>>> #define the constraints and objective function
>>> lin_cons = c_dual_SDP.add_list_of_constraints(
...                  [(AA[i]*AA[i].T | X) <= 1 for i in range(s)])
>>> psd = c_dual_SDP.add_constraint(X>>0)
>>> c_dual_SDP.set_objective('max', cc.T*X*cc)
```

```
>>> print(c_dual_SDP)
Semidefinite Program
  maximize cᵀ·X·c
  over
    5×5 symmetric variable X
  subject to
    ⟨A[i]·A[i]ᵀ, X⟩ ≤ 1 ∀ i ∈ [0...7]
    X ≽ 0
```

```
>>> # solve the problem and retrieve the weights of the optimal design
>>> solution = c_dual_SDP.solve(solver='cvxopt')
>>> mu = [cons.dual for cons in lin_cons] #Lagrangian duals of the linear constraints
>>> mu = cvx.matrix(mu)
>>> w = mu / sum(mu) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[...]
```

(continues on next page)

(continued from previous page)

```
[...]
[ 1.28e-01]
[...]
[ 8.72e-01]
[...]
```

And the optimal positive semidefinite matrix X is:

```
>>> print(X)
[ 5.92e-03  8.98e-03  2.82e-03 -3.48e-02 -1.43e-02]
[ 8.98e-03  1.36e-02  4.27e-03 -5.28e-02 -2.17e-02]
[ 2.82e-03  4.27e-03  1.34e-03 -1.66e-02 -6.79e-03]
[-3.48e-02 -5.28e-02 -1.66e-02  2.05e-01  8.39e-02]
[-1.43e-02 -2.17e-02 -6.79e-03  8.39e-02  3.44e-02]
```

3.5.4 A-optimality (SOCP)

We compute the A-optimal design for the observation matrices `A[i].T` defined in the preamble. 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:

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

```
>>> # create the problem, variables and params
>>> A_primal_SOCP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> Z = [picos.RealVariable('Z[{0}]'.format(i), AA[i].T.size) for i in range(s)]
>>> mu = picos.RealVariable('mu', s)
```

```
>>> #define the constraints and objective function
>>> cone_cons = A_primal_SOCP.add_list_of_constraints(
...                     [abs(Z[i]) <= mu[i] for i in range(s)])
>>> lin_cons = A_primal_SOCP.add_constraint(
...                     picos.sum([AA[i] * Z[i] for i in range(s)]) == 'I')
>>> A_primal_SOCP.set_objective('min', mu.sum)
>>> print(A_primal_SOCP)
Second Order Cone Program
  minimize ∑(mu)
  over
    3×5 real variable Z[i] ∀ i ∈ [0...7]
```

(continues on next page)

(continued from previous page)

```
    8×1 real variable mu
  subject to
    ‖Z[i]‖ ≤ mu[i] ∀ i ∈ [0...7]
    ∑(A[i]·Z[i] : i ∈ [0...7]) = I
```

```
>>> # solve the problem and retrieve the weights of the optimal design
>>> solution = A_primal_SOCP.solve(solver='cvxopt')
>>> w = mu.value
>>> w = w / sum(w) #normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[ 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 tutorial:

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

```
>>> #create the problem, variables and params
>>> D_SOCPual_A=picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> m  = AA[0].size[0]
>>> U  = picos.RealVariable('U',(m,m))
>>> #define the constraints and objective function
>>> cone_cons = D_SOCPual_A.add_list_of_constraints(
...       [abs(AA[i].T*U) <= 1 for i in range(s)])
>>> D_SOCPual_A.set_objective('max', U.tr)
>>> print(D_SOCPual_A)
Second Order Cone Program
  maximize tr(U)
  over
    5×5 real variable U
  subject to
    ‖A[i]ᵀ·U‖ ≤ 1 ∀ i ∈ [0...7]
```

```
>>> # solve the problem and retrieve the weights of the optimal design
>>> solution = D_SOCPual_A.solve(solver='cvxopt')
>>> mu = [cons.dual[0] for cons in cone_cons] # Lagrangian duals of the SOC␣
```

(continues on next page)

(continued from previous page)

```
↪constraints
>>> mu = cvx.matrix(mu)
>>> w = mu / sum(mu) # normalize mu to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[ 2.49e-01]
[ 1.42e-01]
[ 8.51e-02]
[ 1.21e-01]
[ 1.32e-01]
[ 2.70e-01]
```

3.5.5 A-optimality with multiple constraints (SOCP)

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

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

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
>>> A_multiconstraints = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> mu = picos.RealVariable('mu',s)
>>> w  = picos.RealVariable('w',s)
>>> Z  = [picos.RealVariable('Z[{0}]'.format(i), AA[i].T.size)
...                        for i in range(s)]
>>> # define the constraints and objective function
>>> lin_cons0 = A_multiconstraints.add_constraint(
...         picos.sum([AA[i] * Z[i] for i in range(s)]) == 'I')
>>> lin_cons1 = A_multiconstraints.add_constraint(w[:4].sum <= 0.5)
```

(continues on next page)

(continued from previous page)

```
>>> lin_cons2 = A_multiconstraints.add_constraint(w[4:].sum <= 0.5)
>>> cone_cons = A_multiconstraints.add_list_of_constraints(
...         [ abs(Z[i]) **2 <= mu[i] * w[i] for i in range(s)])
>>> A_multiconstraints.set_objective('min', mu.sum)
>>> print(A_multiconstraints)
Quadratically Constrained Program
  minimize ∑(mu)
  over
    3×5 real variable Z[i] ∀ i ∈ [0...7]
    8×1 real variables mu, w
  subject to
    ∑(A[i]·Z[i] : i ∈ [0...7]) = I
    ∑(w[:4]) ≤ 0.5
    ∑(w[4:]) ≤ 0.5
    ‖Z[i]‖² ≤ mu[i]·w[i] ∀ i ∈ [0...7]
```

```
>>> # solve the problem and retrieve the weights of the optimal design
>>> solution = A_multiconstraints.solve(solver='cvxopt')
>>> w = w.value
>>> w = w / sum(w) # normalize w to get the optimal weights
```

The optimal design is:

```
>>> print(w)
[...]
[...]
[ 2.97e-01]
[ 2.03e-01]
[ 6.54e-02]
[ 1.19e-01]
[ 9.02e-02]
[ 2.25e-01]
```

3.5.6 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:

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

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

```
>>> # create the problem, variables and params
>>> A_exact = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> m  = AA[0].size[0]
>>> N  = picos.Constant('N', 20) # number of trials allowed
>>> I = picos.Constant('I', cvx.spmatrix([1]*m,range(m),range(m),(m,m))) #identity␣
↪matrix
>>> Z = [picos.RealVariable('Z[{0}]'.format(i), AA[i].T.size) for i in range(s)]
>>> n = picos.IntegerVariable('n', s)
>>> t = picos.RealVariable('t', s)
```

```
>>> # define the constraints and objective function
>>> cone_cons = A_exact.add_list_of_constraints(
...         [ abs(Z[i])**2 <= n[i] * t[i] for i in range(s)])
>>> lin_cons = A_exact.add_constraint(
...          picos.sum([AA[i]*Z[i] for i in range(s)]) == I)
>>> wgt_cons = A_exact.add_constraint(n.sum <= N)
>>> A_exact.set_objective('min', t.sum)
>>> print(A_exact)
Mixed-Integer Quadratically Constrained Program
  minimize ∑(t)
  over
    8×1 integer variable n
    3×5 real variable Z[i] ∀ i ∈ [0...7]
    8×1 real variable t
  subject to
    ‖Z[i]‖² ≤ n[i]·t[i] ∀ i ∈ [0...7]
    ∑(A[i]·Z[i] : i ∈ [0...7]) = I
    ∑(n) ≤ N
```

```
>>> #solve the problem and display the optimal design
>>> solution = A_exact.solve()
>>> print(n)
[...]
[...]
[ 5.00e+00]
[ 3.00e+00]
[ 2.00e+00]
[ 2.00e+00]
[ 3.00e+00]
[ 5.00e+00]
```

Note: The above output is not validated as we lack an appropriate solver on the build server.

3.5.7 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:

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

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 `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
>>> D_SOCP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> m  = AA[0].size[0]
>>> mm = picos.Constant('m', m)
>>> L = picos.RealVariable('L', (m,m))
>>> V = [picos.RealVariable('V['+str(i)+']', AA[i].T.size) for i in range(s)]
>>> w = picos.RealVariable('w',s)
>>> # additional variable to handle the geometric mean in the objective function
>>> t = picos.RealVariable('t',1)
```

```
>>> # define the constraints and objective function
>>> lin_cons = D_SOCP.add_constraint(picos.sum([AA[i]*V[i] for i in range(s)]) == L)
>>> # L is lower triangular
>>> lowtri_cons = D_SOCP.add_list_of_constraints([L[i,j] == 0
...                  for i in range(m)
...                  for j in range(i+1,m)])
>>> cone_cons = D_SOCP.add_list_of_constraints([abs(V[i]) <= (mm**0.5)*w[i]
...                                              for i in range(s)])
>>> wgt_cons = D_SOCP.add_constraint(w.sum <= 1)
>>> geomean_cons = D_SOCP.add_constraint(t <= picos.geomean(picos.maindiag(L)))
>>> D_SOCP.set_objective('max',t)
```

```
>>> #solve the problem and display the optimal design
>>> print(D_SOCP)
Optimization Problem
  maximize t
  over
    1×1 real variable t
    3×5 real variable V[i] ∀ i ∈  [0...7]
    5×5 real variable L
    8×1 real variable w
```

(continues on next page)

(continued from previous page)

```
  subject to
    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^(1/2)·w[i] ∀ i ∈ [0...7]
    ∑(w) ≤ 1
    geomean(maindiag(L)) ≥ t
```

```
>>> solution = D_SOCP.solve(solver='cvxopt')
>>> print(w)
[...]
[...]
[ 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
>>> D_exact = picos.Problem()
>>> L = picos.RealVariable('L',(m,m))
>>> V = [picos.RealVariable('V['+str(i)+']',AA[i].T.size) for i in range(s)]
>>> T = picos.RealVariable('T', (s,m))
>>> n = picos.IntegerVariable('n', s)
>>> N = picos.Constant('N', 20)
>>> # additional variable to handle the geomean inequality
>>> t = picos.RealVariable('t',1)
```

```
>>> # define the constraints and objective function
>>> lin_cons = D_exact.add_constraint(
...         picos.sum([AA[i]*V[i] for i in range(s)]) == L)
>>> # L is lower triangular
>>> lowtri_cons = D_exact.add_list_of_constraints([L[i,j] == 0
...                                 for i in range(m)
...                                 for j in range(i+1,m)])
>>> cone_cons = D_exact.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)])
>>> lin_cons2 = D_exact.add_list_of_constraints([T[:,k].sum <= 1
...                       for k in range(m)])
>>> wgt_cons = D_exact.add_constraint(n.sum <= N)
>>> geomean_cons = D_exact.add_constraint(t <= picos.geomean(picos.maindiag(L)))
>>> D_exact.set_objective('max',t)
>>> print(D_exact)
Mixed-Integer Optimization Problem
  maximize t
  over
    8×1 integer variable n
    1×1 real variable t
    3×5 real variable V[i] ∀ i ∈ [0...7]
    5×5 real variable L
    8×5 real variable T
  subject to
```

(continues on next page)

(continued from previous page)

```
    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,j] ∀ (i,j) ∈
      zip([0,0,...,7,7],[0,1,...,3,4])
    ∑(T[:,i]) ≤ 1 ∀ i ∈ [0...4]
    ∑(n) ≤ N
    geomean(maindiag(L)) ≥ t
```

```
>>> #solve the problem and display the optimal design
>>> solution = D_exact.solve()
>>> print(n)
[...]
[...]
[ 5.00e+00]
[ 1.00e+00]
[...]
[ 1.00e+00]
[ 6.00e+00]
[ 7.00e+00]
```

Note: The above output is not validated as we lack an appropriate solver on the build server.

3.5.8 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
>>> D_MAXDET = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> m  = AA[0].size[0]
>>> w = picos.RealVariable('w', s, lower=0)
>>> t = picos.RealVariable('t', 1)
>>> # constraint and objective
>>> wgt_cons = D_MAXDET.add_constraint(w.sum <= 1)
>>> Mw = picos.sum([w[i] * AA[i] * AA[i].T for i in range(s)])
>>> detrootn_cons = D_MAXDET.add_constraint(t <= picos.DetRootN(Mw))
>>> D_MAXDET.set_objective('max', t)
```

```
>>> print(D_MAXDET)
Optimization Problem
  maximize t
  over
    1×1 real variable t
    8×1 real variable w (bounded below)
  subject to
    ∑(w) ≤ 1
    det(∑(w[i]·A[i]·A[i]ᵀ : i ∈ [0...7]))^(1/5) ≥ t
```

```
>>> #solve and display
>>> solution = D_MAXDET.solve(solver='cvxopt')
>>> print(w)
[ ...]
[ ...]
[ 2.27e-01]
[ 3.38e-02]
[ 1.65e-02]
[ 5.44e-02]
[ 3.18e-01]
[ 3.51e-01]
```

3.5.9 General Phi_p optimal design (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 $(-\infty, 1]$:

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

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
>>> P0dot2_SDP  = picos.Problem()
>>> Pminus3_SDP = picos.Problem()
>>> AA = [picos.Constant('A[{0}]'.format(i), Ai)
...       for i, Ai in enumerate(A)] # each AA[i].T is a 3 x 5 observation matrix
>>> s  = len(AA)
>>> m  = AA[0].size[0]
>>> w = picos.RealVariable('w', s, lower=0)
>>> t = picos.RealVariable('t', 1)
```

```
>>> # constraint and objective
>>> wgt02_cons = P0dot2_SDP.add_constraint(w.sum <= 1)
>>> wgtm3_cons = Pminus3_SDP.add_constraint(w.sum <= 1)
```

```
>>> Mw = picos.sum([w[i]*AA[i]*AA[i].T for i in range(s)])
```

```
>>> tracep02_cons = P0dot2_SDP.add_constraint(t <= picos.PowerTrace(Mw, 0.2))
>>> P0dot2_SDP.set_objective('max', t)
```

```
>>> tracepm3_cons = Pminus3_SDP.add_constraint(t >= picos.PowerTrace(Mw, -3))
>>> Pminus3_SDP.set_objective('min', t)
```

```
>>> # p=0.2
>>> print(P0dot2_SDP)
```

(continues on next page)

(continued from previous page)

```
Optimization Problem
  maximize t
  over
    1×1 real variable t
    8×1 real variable w (bounded below)
  subject to
    ∑(w) ≤ 1
    tr(∑(w[i]·A[i]·A[i]ᵀ : i ∈ [0...7])^(1/5)) ≥ t
```

```
>>> #solve and display
>>> solution = P0dot2_SDP.solve(solver='cvxopt')
>>> print(w)
[ ...]
[ ...]
[ 2.06e-01]
[ ...]
[ ...]
[ 9.20e-03]
[ 4.08e-01]
[ 3.77e-01]
```

```
>>> # p=-3
>>> print(Pminus3_SDP)
Optimization Problem
  minimize t
  over
    1×1 real variable t
    8×1 real variable w (bounded below)
  subject to
    ∑(w) ≤ 1
    tr(∑(w[i]·A[i]·A[i]ᵀ : i ∈ [0...7])^(-3)) ≤ t
>>> solution = Pminus3_SDP.solve(solver='cvxopt')
>>> print(w)
[ ...]
[ ...]
[ 2.48e-01]
[ 1.66e-01]
[ 1.08e-01]
[ 1.41e-01]
[ 7.83e-02]
[ 2.60e-01]
```

3.5.10 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.

Warning: This part of the documentation has not been touched for a while. It might be incomplete, reference deprecated functions or make a claim that does not apply to the latest version of PICOS any more. On the bright side, code listings are validated and still work. Watch your step!

3.6 Additional constraints

This section introduces additional expression and constraint types that didn't fit into the tutorial. Again, let us import PICOS:

```
>>> import picos as pc
```

We replicate some of the variables and data used in the tutorial:

```
>>> from picos import Constant, RealVariable
>>> t = RealVariable("t")
>>> x = RealVariable("x", 4)
>>> Y = RealVariable("Y", (2, 4))
>>> alpha = Constant("α", 23)
>>> A = [Constant("A[{}]".format(i), range(i, i + 8), (2, 4)) for i in range(5)]
```

This time, Z and b are lists:

```
>>> Z = [RealVariable("Z[{0}]".format(i), (4,2)) for i in range(5)]
>>> b = ([0, 2, 0, 3], [1, 1, 0, 5], [-1, 0, 2, 4], [0, 0, -2, -1],
...      [1, 1, 0, 0])
>>> b = [Constant("b[{}]".format(i), b[i]) for i in range(len(b))]
```

3.6.1 Graph flow constraints

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 = pc.Problem()
>>> # Adding the flow variables
>>> f={}
>>> for e in G.edges():
...     f[e]=pb.add_variable('f[{},{}]'.format(e[0], e[1]), 1)
>>> # A variable for the value of the flow
>>> F = pb.add_variable('F',1)
>>> # Creating the flow constraint
>>> flowCons = pb.add_constraint(pc.flow_Constraint(
...     G, f, 'S', 'T', F, capacity='capacity', graphName='G'))
>>> pb.set_objective('max',F)
>>> sol = pb.solve(solver='cvxopt')
>>> flow = {key: var.value for key, var in f.items()}
```

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

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

flowCons = pc.flow_Constraint(
    G, f, source='S', sink=['T1','T2'], capacity='capacity',
    flow_value=[F1, F2], graphName='G')

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

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

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

```
Linear Program
  maximize F1 + F2
  over
    1×1 real variables F1, F2, f[A,T1], f[B,T2], f[S,A], f[S,B]
  subject to
    Feasible S-(T1,T2)-flow in G has values 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.

3.6.2 Second Order Cone constraints

Warning: This section in particular is outdated: The only direct way to create rotated second order cone constraints is now via the *rsoc* set-generating function. If you input such a constraint as below, then you will receive either a convex or a conic quadratic constraint. The former is handled depending on the solver used. The latter will be transformed either to a rotated conic constraint (which implicitly adds additional constraints that are part of the rotated second order cone definition) or it will remain nonconvex quadratic, depending on an option.

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
>>> abs(Y+Z[0].T) < t+alpha
<9×1 SOC Constraint: ‖Y + Z[0]ᵀ‖ ≤ t + α>
>>> # Rotated SOC constraint
>>> abs(Z[1][:,0])**2 < (2*t-alpha)*(x[2]-x[-1])
<Conic Quadratic Constraint: ‖Z[1][:,0]‖² ≤ (2·t - α)·(x[2] - x[-1])>
>>> # t**2 is internally represented as the squared norm of [t]
>>> t**2 < alpha + t
<Squared Scalar Constraint: t² ≤ α + t>
>>> # 1 is understood as the squared norm of [1]
>>> 1 < (t-1)*(x[2]+x[3])
<Conic Quadratic Constraint: (t - 1)·(x[2] + x[3]) ≥ 1>
```

3.6.3 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:

```
>>> pc.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

```
>>> pc.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 = pc.Problem()
>>> X = sdp.add_variable('X',(4,4),vtype='symmetric')
>>> C = sdp.add_constraint(X >> 0)
>>> print(sdp)
Feasibility Problem
  find an assignment
  for
    4×4 symmetric variable X
  subject to
    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 enforced that $A - B$ is symmetric. How the constraint is passed then depends on the solver, for instance it could be that the lower or upper triangular part is ignored. You can add a constraint of the form `A - B == (A - B).T` to enforce symmetry.

3.6.4 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 `geomean`:

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

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>
```

Such a power constraint will be reformulated as a geometric mean inequality when the problem is solved, which in turn will be translated to conic inequalities.

3.6.5 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]>
>>> t**-0.5 > x[0]
Traceback (most recent call last):
  ...
TypeError: Cannot lower-bound a nonconcave (trace of) power.
```

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

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

```
>>> pc.tracepow(x[0] * A[0]*A[0].T + x[2] * A[2]*A[2].T, 2.5) <= 3
<Trace of Power Constraint: tr((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 $\operatorname{trace}\ X^p$) 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 $\operatorname{trace}(MX^p) \geq t$ with SDPs, where $M \succeq 0$, see *[2]*.

```
>>> pc.tracepow(X, 0.6666, coef = A[0].T*A[0]+"I") >= t
<Trace of Scaled Power Constraint: tr((A[0]ᵀ·A[0] + I)·X^(2/3)) ≥ t>
```

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

3.6.6 Inequalities involving generalized p-norm

Inequalities of the form $\|x\|_p \leq t$ can be entered by using the function *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 $p \leq 1$.

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

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 *norm* .

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

3.6.7 Inequalities involving the nth root of a determinant

The function *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 < pc.detrootn(M)
<n-th Root of a Determinant Constraint: det(M)^(1/5) ≥ t>
```

Warning: Note that when a constraint of the form `t < pc.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.

3.6.8 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 *ball*
- The standard simplex (scaled by a factor γ) $\{x \geq 0 : \sum_i x_i \leq r\}$ can be constructed with the function *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 *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 << pc.simplex(1)
<Unit Simplex Constraint: x ∈ {x ≥ 0 : ∑(x) ≤ 1}>
>>> x << pc.truncated_simplex(2)
<Box-Truncated Simplex Constraint: x ∈ {0 ≤ x ≤ 1 : ∑(x) ≤ 2}>
>>> x << pc.truncated_simplex(2,sym=True)
<Box-Truncated 1-norm Ball Constraint: x ∈ {-1 ≤ x ≤ 1 : ∑(|x|) ≤ 2}>
>>> x << pc.ball(3)
<5×1 SOC Constraint: ‖x‖ ≤ 3>
>>> pc.ball(2,'inf') >> x
<Maximum Norm Constraint: ‖x‖_max ≤ 2>
>>> x << pc.ball(4,1.5)
<Vector p-Norm Constraint: ‖x‖_(3/2) ≤ 4>
```

3.6.9 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.

CHAPTER

FOUR

PICOS FOR QUANTUM INFORMATION SCIENCE

PICOS was among the first convex optimization interfaces to natively support Hermitian semidefinite programming and subsystem manipulation operations such as the partial trace and partial transpose, which were implemented with feedback from the QIS community. This note outlines the features most relevant for the field and links to examples.

4.1 Cheat sheet

Table 1: Complex expression manipulation

on paper	in picos
$A \in \mathbb{C}^{m \times n}$	`A = pc.ComplexVariable("A", (m, n))`
$A = B + iC$	`A = B + 1j*C`
$\overline{A}$	`A.conj`
$A^\dagger$	`A.H`
$\Re(A)$	`A.real`
$\Im(A)$	`A.imag`
$\frac{1}{2}\left(A + A^\dagger\right)$	`A.opreal` / `A.hermitianized`
$\frac{1}{2i}\left(A - A^\dagger\right)$	`A.opimag`
$\langle\phi\|\psi\rangle$	`phi.H * psi` / `(phi | psi)`
$\|\phi\rangle\langle\psi\|$	`phi * psi.H`

Table 2: Hermitian semidefinite programming

on paper	in picos
$\rho \in \mathbb{S}^n$	`rho = pc.HermitianVariable("ρ", n)`
$\rho \succeq 0$	`rho >> 0`
$\rho \succeq I$	`rho >> 1` / `rho >> pc.I(n)`
$\mathrm{Tr}(\rho) = 1$	`rho.tr == 1`
$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \succeq 0$	`pc.block([[A, B], [C, D]]) >> 0`

Table 3: Schatten norms

on paper	in picos	note / aka
$\|A\|_1 = \mathrm{Tr}\left(\sqrt{A^\dagger A}\right)$	`pc.NuclearNorm(A)`	*trace norm*
$\|A\|_\infty = \sqrt{\lambda_{\max}(A^\dagger A)}$	`pc.SpectralNorm(A)`	$\lambda_{\max}(A)$ for $A \in \mathbb{H}^n$

Table 4: Subsystem manipulation (partial trace, partial transpose, realignment)

on paper	in picos	note / docs
$A = B \otimes C$	`A = B @ C`	
$A_1 \otimes \cdots \otimes \operatorname{Tr}(A_i) \otimes \cdots \otimes A_n$	`A.partial_trace([i-1], shapes)`	*partial_trace*
$A_1 \otimes \cdots \otimes A_i^T \otimes \cdots \otimes A_n$	`A.partial_tranpose([i-1], shapes)`	*partial_transpose*
$A_{ij \,\mapsto\, ji} = A^T$	`A.reshuffled("ji")`	*reshuffled*
$A_{ijkl \,\mapsto\, kjil} = \mathrm{T}_1(A)$	`A.reshuffled("kjil")`	*reshuffled*
$\operatorname{Tr}_1(A),\ \ldots,\ \operatorname{Tr}_4(A),\ \operatorname{Tr}_{\text{last}}(A)$	`A.tr0`, …, `A.tr3`, `A.trl`	$A \in \mathbb{H}^2 \otimes \cdots \otimes \mathbb{H}^2$
$\mathrm{T}_1(A),\ \ldots,\ \mathrm{T}_4(A),\ \mathrm{T}_{\text{last}}(A)$	`A.T0`, …, `A.T3`, `A.Tl`	$A \in \mathbb{H}^2 \otimes \cdots \otimes \mathbb{H}^2$

(Tr_i and T_i denote the partial trace and transpose of the i-th 2×2 subsystem, counted from zero)

4.2 Hermitian semidefinite programming

PICOS makes use of the following identity to allow standard solvers to deal with hermitian LMIs:

$$A \succeq 0 \qquad \iff \qquad \begin{bmatrix} \Re(A) & \Im(A) \\ -\Im(A) & \Re(A) \end{bmatrix} \succeq 0$$

Hermitian variables are vectorized such that $\rho \in \mathbb{S}^n$ is passed to solvers via n^2 real scalar variables. Alternatively, the QICS solver is able to directly handle hermitian variables.

4.3 Quantum relative entropy programming

As of version 2.5.0, PICOS supports solving quantum relative entropy programs with the solver QICS. A list of new expressions supported by PICOS and QICS is summarized below.

Table 5: Quantum entropies and non-commutative perspectives

on paper	in picos	docs
$S(X) = -\operatorname{Tr}(X\log(X))$	`pc.quantentr(X)`	*QuantumEntropy*
$S(X\|Y) = \operatorname{Tr}(X\log(X) - X\log(Y))$	`pc.quantrelentr(X, Y)`	*NegativeQuantumEntropy*
$S(X) - S(\operatorname{Tr}_i(X))$	`pc.quantcondentr(X, [i-1], shapes)`	*QuantumConditionalEntropy*
$S(\mathcal{G}(X)\|\mathcal{Z}(\mathcal{G}(X)))$	`pc.quantkeydist(X, [i-1], shapes, K_list)`	*QuantumKeyDistribution*
$P_{\log}(X,Y) = X^{1/2}\log(X^{1/2}Y^{-1}X^{1/2})X^{1/2}$	`pc.oprelentr(X, Y)`	*OperatorRelativeEntropy*
$X \#_t Y = X^{1/2}(X^{1/2}Y^{-1}X^{1/2})^t X^{1/2}$	`pc.mtxgeomean(X, Y, t)`	*MatrixGeometricMean*

Some examples for how to solve quantum relative entropy programs using PICOS can be found *here*. Note that these functions are supported for both real symmetric and complex hermitian matrices X and Y.

4.4 Examples and exercises

- *Fidelity between operators*
- *Quantum relative entropy programs*
- Quantum channel discrimination (exercise on Binder)

4.5 Course material

Jupyter notebooks for a hands-on workshop on practical semidefinite programming aimed at quantum information students are available on GitLab. The fourth notebook is based on [*2*], which also comes with Python/PICOS notebooks.

4.6 Recent articles

The following are peer-reviewed articles relating to quantum information that cite PICOS and were published within the last four years (last update: October 2024).

- Vikesh Siddhu and John Smolin.

  Maximum expectation of observables with restricted purity states.

  Quantum 8, 2024. [pdf] [doi] [arXiv]

- Aby Philip, Soorya Rethinasamy,Vincent Russo, and M. Wilde.

  Schrödinger as a quantum programmer: estimating entanglement via steering.

  Quantum 8, 2024. [pdf] [doi] [arXiv]

- Piotr Mironowicz.

  Semi-definite programming and quantum information.

  Journal of Physics A: Mathematical and Theoretical 57, 2024. [pdf] [doi] [arXiv]

- Yu Shi and Edo Waks.

  Error metric for non-trace-preserving quantum operations.

  Physical Review A 108, 2023. [pdf] [doi] [arXiv]

- Vincent Russo and Jamie Sikora.

  Inner products of pure states and their antidistinguishability.

  Physical Review A 107, 2023. [pdf] [doi] [arXiv] [code]

- Armin Tavakoli, Alejandro Pozas-Kerstjens, Ming-Xing Luo, and Marc-Olivier Renou.

  Bell nonlocality in networks.

  Reports on Progress in Physics 85, 2022. [pdf] [doi] [arXiv]

- Feng-Jui Chan et al.

  Maxwell's two-demon engine under pure dephasing noise.

  Physical Review A 106, 2022. [pdf] [doi] [arXiv]

- Viktor Nordgren et al.

  Certifying emergent genuine multipartite entanglement with a partially blind witness.

  Physical Review A 106, 2022. [pdf] [doi] [arXiv]

- Vikesh Siddhu and Sridhar Tayur.

  Five starter pieces: quantum information science via semidefinite programs.

  Tutorials in Operations Research, 2022. [pdf] [doi] [arXiv]

- Ulysse Chabaud, Pierre-Emmanuel Emeriau, and Frédéric Grosshans.

  Witnessing Wigner negativity.

  Quantum 5, 2021. [pdf] [doi] [arXiv] [code]

4.7 Ncpol2sdpa

Ncpol2sdpa [*1*] exposes SDP relaxations of (non-commutative) polynomial optimization problems as PICOS problem instances, see here.

4.8 References

1. Peter Wittek. Algorithm 950: Ncpol2sdpa—sparse semidefinite programming relaxations for polynomial optimization problems of noncommuting Variables. *ACM Transactions on Mathematical Software*, 41(3), 21, 2015. DOI: 10.1145/2699464. arXiv: 1308.6029.
2. Vikesh Siddhu and Sridhar Tayur. Five starter pieces: quantum information science via semi-definite programs. *Tutorials in Operations Research*, 2022. DOI: 10.1287/educ.2022.0243. arXiv: 2112.08276.

CHAPTER

FIVE

USAGE NOTES

5.1 NumPy and SciPy

As a lightweight computer algebra system, PICOS sits one level above numerics libraries such as NumPy and SciPy and acts in concert with them. Let's define a variable and some data:

```
>>> import picos, numpy, scipy.sparse
>>> x = picos.RealVariable("x", 4)
>>> N = numpy.reshape(range(16), (4, 4))
>>> type(N)
<class 'numpy.ndarray'>
>>> S = scipy.sparse.spdiags(range(4), 0, 4, 4)
>>> type(S)
<class 'scipy.sparse._dia.dia_matrix'>
```

Taking input from NumPy or SciPy

PICOS also allows loading of NumPy and SciPy data on the fly, with one caveat to watch out for:

```
>>> x.T*N
<1×4 Real Linear Expression: xᵀ·[4×4]>
>>> N*x
<4×1 Real Linear Expression: [4×4]·x>
>>> x.T*S
<1×4 Real Linear Expression: xᵀ·[4×4]>
>>> S*x
Traceback (most recent call last):
    [...]
picos.valuable.NotValued: Mutable x is not valued.
```

The last command fails as SciPy sparse matrices do not currently respect the __array_priority__ attribute, so that SciPy tries to load `x` as an array as opposed to conceding the operation to PICOS like NumPy does. You can fix this behavior as follows:

```
>>> picos.patch_scipy_array_priority()
>>> S*x
<4×1 Real Linear Expression: [4×4]·x>
```

Note that this monkey-patches SciPy, so that applications importing your code calling *patch_scipy_array_priority* will also see a patched version of SciPy.

Returning NumPy or SciPy data as output

PICOS uses CVXOPT as a numerics backend and thus outputs numeric values as CVXOPT (sparse) matrices or Python scalar types by default:

```
>>> x.value = range(4)
>>> x.value
<4x1 matrix, tc='d'>
>>> type(x.value)
<class 'cvxopt.base.matrix'>
```

However, all objects that can be valued, in particular expressions and problem instances, also offer properties to query that value as a NumPy type, namely *np* and *np2d*:

```
>>> x.np  # Returns a NumPy scalar, 1D, or 2D array.
array([0., 1., 2., 3.])
>>> type(x.np)
<class 'numpy.ndarray'>
>>> x.np.shape
(4,)
>>> x.np2d  # Always returns a 2D array.
array([[0.],
       [1.],
       [2.],
       [3.]])
>>> x.np2d.shape
(4, 1)
```

For SciPy, the *sp* property returns a sparse matrix whenever the data stored by PICOS internally is sparse and a NumPy 2D array otherwise:

```
>>> I = picos.I(3)
>>> print(I)
[ 1.00e+00     0         0    ]
[    0      1.00e+00     0    ]
[    0         0      1.00e+00]
>>> type(I.sp)
<class 'scipy.sparse._csc.csc_matrix'>
>>> J = picos.J(3, 3)
>>> print(J)
[ 1.00e+00  1.00e+00  1.00e+00]
[ 1.00e+00  1.00e+00  1.00e+00]
[ 1.00e+00  1.00e+00  1.00e+00]
>>> type(J.sp)
<class 'numpy.ndarray'>
```

A full list of methods for returning values in different formats can be found in the documentation of the `Valuable` base class.

5.2 Matrix Slicing

Affine matrix expressions form the core of PICOS' modeling toolbox: All `constant` and `variable` expressions that you enter, including integral variables, and any linear combination of these objects, are stored as instances of the multidimensional `ComplexAffineExpression` or its real subclass `AffineExpression`. Their common base class `BiaffineExpression` implements plenty of algebraic operations to combine and modify your initial expressions to yield the desired statements. One of these operations is slicing, denoted by `A[·]` for an affine expression `A`.

Preliminaries

Unlike in NumPy, all multidimensional expressions are strictly matrices. In particular, there are no flat arrays but only row and column vectors, and any scalar expression is also a 1×1 matrix. PICOS does not support tensors or higher order expressions, but it does support the Kronecker product as well as `partial trace` and `partial transposition` operations to enable some of the optimization problems naturally defined on tensors. If you enter data in the form of a flat array (e.g. a Python `list` or a NumPy `ndarray` with one axis), it will be read as a column vector.

In PICOS, all indices start from zero.

Slicing modes

PICOS has two modes for slicing: *Arbitrary Access* and *Proper Slicing*.

Arbitrary Access lets you select individual elements from a vector or matrix expression and put them in a column vector in the desired order. `Transposition`, `reshaping` and `broadcasting` can then be used to put the selection into the desired shape. Arbitrary Access has the form `A[·]` where · stands for an integer, a Python `slice`, a flat collection of integers such as a `list`, or a dictionary storing sparse index pairs.

Proper Slicing refers to selecting certain rows and columns of a matrix, and forming a new matrix where all elements that are not selected are removed. It has the form `A[·,·]` where each · stands for an integer, a `slice`, or a flat collection of integers.

To demonstrate the different possibilities, we use a constant 5×5 expression:

```
>>> from picos import Constant
>>> A = Constant("A", range(25), (5,5))
>>> A
<5×5 Real Constant: A>
>>> print(A)
[ 0.00e+00  5.00e+00  1.00e+01  1.50e+01  2.00e+01]
[ 1.00e+00  6.00e+00  1.10e+01  1.60e+01  2.10e+01]
[ 2.00e+00  7.00e+00  1.20e+01  1.70e+01  2.20e+01]
[ 3.00e+00  8.00e+00  1.30e+01  1.80e+01  2.30e+01]
[ 4.00e+00  9.00e+00  1.40e+01  1.90e+01  2.40e+01]
```

5.2.1 Arbitrary Access

By integer

If a single integer or a single flat collection of integers is given, then these indices refer to the column-major vectorization of the matrix, represented by the order of the numbers in the demonstration matrix `A`.

The most common case is selecting a single element via an integer index:

```
>>> A[0]  # Select the first element as a scalar expression.
<1×1 Real Constant: A[0]>
>>> print(A[0])  # Print its value.
0.0
>>> print(A[7])  # The value of the eighth element.
7.0
>>> # Negative indices are counted from the rear; -1 refers to the last element:
>>> print(A[-1])
24.0
```

By slice

Python slices allow you to compactly specify a structured sequence of elements to extract. A Python slice has the form `a:b` or `a:b:s` with a the inclusive start index, b the exclusive stop index and s a step size. Negative a and b, as in the integer index case, are counted from the rear, while a negative step size reverses the order. All of a, b and s may be omitted. Then, the defaults are

$$\begin{aligned} s &= 1, \\ a &= \begin{cases} 0, & \text{if } s > 0, \\ \operatorname{len}(A) - 1, & \text{if } s < 0, \end{cases} \\ b &= \begin{cases} \operatorname{len}(A), & \text{if } s > 0, \\ \textbf{None}, & \text{if } s < 0. \end{cases} \end{aligned}$$

Note the `None` in the statement above: When going backwards, this special token is the only way to stop at the first element with index 0 as -1 refers to the last element. For example, the first two elements in reverse order are selected via the slice `1:None:-1` or just `1::-1`.

```
>>> A[:2]  # The first two elements as a column vector.
<2×1 Real Constant: A[:2]>
>>> print(A[:2])
[ 0.00e+00]
[ 1.00e+00]
>>> print(A[1::-1])  # The first two elements reversed (indices 1 and 0).
[ 1.00e+00]
[ 0.00e+00]
>>> print(A[-2:])  # The last two elements.
[ 2.30e+01]
[ 2.40e+01]
>>> print(A[2:7].T)  # The third to seventh element (transposed).
[ 2.00e+00  3.00e+00  4.00e+00  5.00e+00  6.00e+00]
>>> print(A[2:7:2].T)  # As before, but with a step size of 2.
[ 2.00e+00  4.00e+00  6.00e+00]
```

You could use this to vectorize A in column-major order, but `A.vec` is both individually faster and has its result cached:

```
>>> A[:].equals(A.vec)
True
>>> A.vec is A.vec  # Cached.
True
>>> A[:] is A[:]  # Computed again as new expression.
False
```

By integer sequence

By providing a list or a similar vector of integers, you can select arbitrary elements in any order, including duplicates:

```
>>> print(A[[0,1,0,1,-1]])
[ 0.00e+00]
[ 1.00e+00]
[ 0.00e+00]
[ 1.00e+00]
[ 2.40e+01]
```

Note that you cannot provide a tuple instead of a list, as `A[(·,·)]` is understood by Python as `A[·,·]` (see *Proper Slicing*). Any other object that the function *load_data* with `typecode="i"` loads as an integer row or column vector works, including integral NumPy arrays.

By sparse index pair dictionary

If you provide a dictionary with exactly two keys that can be compared via < and whose values are integer sequences of same length (anything recognized by *load_data* as an integer vector), PICOS interprets the sequence corresponding to the smaller key as row indices and the sequence corresponding to the greater key as the corresponding column indices:

```
>>> print(A[{"x": range(3), "y": [1]*3}])  # Select (0,1), (1,1) and (2,1).
[ 5.00e+00]
[ 6.00e+00]
[ 7.00e+00]
>>> print(A[{"y": range(3), "x": [1]*3}])  # Transposed selection, as "x" < "y".
[ 1.00e+00]
[ 6.00e+00]
[ 1.10e+01]
```

You could use this to extract the main diagonal of A, but `A.maindiag` is both individually faster and has its result cached:

```
>>> indices = dict(enumerate([range(min(A.shape))]*2))
>>> indices
{0: range(0, 5), 1: range(0, 5)}
>>> A[indices].equals(A.maindiag)
True
>>> A.maindiag is A.maindiag  # Cached.
True
>>> A[indices] is A[indices]  # Computed again as new expression.
False
```

5.2.2 Proper Slicing

If you provide not one but two integers, slices, or integer sequences separated by a comma or given as a tuple, then they are understood as row and column indices, respectively. Unlike when providing a sparse index pair by dictionary, these indices select *entire* rows and columns and PICOS returns the matrix of all elements that are selected twice (both by row and by column):

```
>>> print(A[1,2])  # The single element at (1,2) (second row, third column).
11.0
>>> print(A[0,:])  # The first row of the matrix.
[ 0.00e+00  5.00e+00  1.00e+01  1.50e+01  2.00e+01]
```

(continues on next page)

(continued from previous page)

```
>>> print(A[range(3),-1])  # The first three elements of the last column.
[ 2.00e+01]
[ 2.10e+01]
[ 2.20e+01]
>>> print(A[[0,1],[0,1]])  # The first second-order principal submatrix.
[ 0.00e+00  5.00e+00]
[ 1.00e+00  6.00e+00]
>>> print(A[1:-1,1:-1])  # Cut away the outermost pixels of an image.
[ 6.00e+00  1.10e+01  1.60e+01]
[ 7.00e+00  1.20e+01  1.70e+01]
[ 8.00e+00  1.30e+01  1.80e+01]
>>> print(A[::2,::2])  # Sample every second element.
[ 0.00e+00  1.00e+01  2.00e+01]
[ 2.00e+00  1.20e+01  2.20e+01]
[ 4.00e+00  1.40e+01  2.40e+01]
```

You can even select the entire matrix to effectively create a copy of it, though this is discouraged as expressions are supposed to be immutable so that reusing an expression in multiple places is always safe.

```
>>> A[:,:].equals(A)
True
>>> A[:,:] is A
False
```

We refer to this as proper slicing because you cut out the rows that you want, throwing away the rest, then cut the desired columns out from the remainder. It's like cutting a square cake except that you can also duplicate the pieces!

Note: In NumPy, `A[[0,1],[0,1]]` would create a flat array with the elements `A[0,0]` and `A[1,1]` while PICOS creates a submatrix from the first two rows and columns as in the example above. If you want to mirror NumPy's behavior in PICOS, see *By sparse index pair dictionary*.

5.3 Dual Values

Picos typically reformulates optimization problems as conic programs of the form

$$\begin{array}{ll} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} & \mathbf{c}^T\mathbf{x}+\gamma \\ \text{subject to} & A_i(\mathbf{x}) \quad \succeq_{K_i} \mathbf{b}_i,\ \forall i\in I, \end{array}$$

where each A_i is a linear map from $\mathbb{R}^n$ to a linear space containing the cone K_i, and the generalized conic inequality $\mathbf{x} \succeq_K \mathbf{y}$ means $\mathbf{x}-\mathbf{y}\in K$ for a cone K. For the sake of compactness, we allow generalized inequalities over the trivial cone $K_{eq}=\{\mathbf{0}\}$, such that $A\mathbf{x}\succeq_{K_{eq}}\mathbf{b}$ represents an equality constraint $A\mathbf{x}=\mathbf{b}$.

The dual conic problem can be written as follows:

$$\begin{array}{ll} \text{maximize} & \sum_{i\in I}\mathbf{b}_i^T\mathbf{y}_i+\gamma \\ \text{subject to} & \sum_{i\in I}A_i^*(\mathbf{y}_i)=\mathbf{c}, \\ & \mathbf{y}_i\succeq_{K_i^*}0,\ \forall i\in I, \end{array}$$

where A^* denotes the adjoint operator of A and K^* denotes the the dual cone of K (see the note below for a list of cones that are supported in PICOS, together with their dual).

After an optimization problem has been solved, we can query the optimal dual variable $y_i\in K_i^*$ of a conic constraint `con` over the cone K_i with its `dual` attribute, i.e., `con.dual`.

When an optimization problem `P` can be reformulated to a conic program `C` of the above form by PICOS, we can use its `dual` attribute to return a `Problem` object `D=P.dual` which contains the dual conic program of `C`. It is also possible to solve `P` via its dual by using the *dualize* option: This passes problem `D` to the solver, and the optimal primal and dual variables of `P` will be retrieved from the optimal solution of `D`.

5.3.1 Supported cones and their dual

PICOS can provide dual information for problems involving the following cones:

Trivial cone

The trivial cone $K_{eq} = \{\mathbf{0}\} \subset \mathbb{R}^n$, whose dual cone is the entire space $K_{eq}^* = \mathbb{R}^n$. This means that the dual variable $\mathbf{y}$ for an equality constraint is unconstrained.

Nonnegative Orthant

The nonnegative orthant $\mathbb{R}_+^n$ is self dual: $(\mathbb{R}_+^n)^* = \mathbb{R}_+^n$. Therefore the dual variable for a set of linear inequalities is a vector $\mathbf{y} \geq \mathbf{0}$.

Lorentz Cone

The *Lorentz cone* $\mathcal{Q}^n = \{(t, \mathbf{x}) \in \mathbb{R} \times \mathbb{R}^{n-1} : \|\mathbf{x}\| \leq t\}$, which is used to model second-order cone inequalities, is self-dual: $(\mathcal{Q}^n)^* = \mathcal{Q}^n$. This means that the dual variable for a second order cone inequality of the form $\|A\mathbf{x} - \mathbf{b}\| \leq \mathbf{h}^T\mathbf{x} - g \iff \begin{bmatrix} \mathbf{h}^T \\ A \end{bmatrix} \mathbf{x} \succeq_{\mathcal{Q}^n} \begin{bmatrix} g \\ \mathbf{b} \end{bmatrix}$ is a vector of the form $[\lambda, \mathbf{z}^T]^T$ such that $\|\mathbf{z}\| \leq \lambda$.

Rotated Second-order Cone

The (widened or narrowed) *rotated second order cone* is $\mathcal{R}_p^n = \{(u, v, \mathbf{x}) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n-2} : \|\mathbf{x}\|^2 \leq p \cdot u \cdot v,\ u, v \geq 0\}$ for some $p > 0$, and its dual cone is $(\mathcal{R}_p^n)^* = \mathcal{R}_{4/p}^n$. In particular, $\mathcal{R}_p^n$ is self-dual for $p = 2$. For example, the dual variable for the constraint $\|A\mathbf{x} - \mathbf{b}\|^2 \leq (\mathbf{h}^T\mathbf{x} - g)(\mathbf{e}^T\mathbf{x} - f)$ with $(\mathbf{h}^T\mathbf{x} - g) \geq 0$ and $(\mathbf{e}^T\mathbf{x} - f) \geq 0$, i.e., $\begin{bmatrix} \mathbf{h}^T \\ \mathbf{e}^T \\ A \end{bmatrix} \mathbf{x} \succeq_{\mathcal{R}_1^n} \begin{bmatrix} g \\ f \\ \mathbf{b} \end{bmatrix}$ is a vector of the form $[\alpha, \beta, \mathbf{z}^T]^T$ such that $\|\mathbf{z}\|^2 \leq 4\alpha\beta$;

Positive Semi-definite Cone

The positive semidefinite cone $\mathbb{S}_+^n$ is self dual: $(\mathbb{S}_+^n)^* = \mathbb{S}_+^n$. This means that the dual variable for a linear matrix inequality $\sum_i x_i M_i \succeq M_0$ is a positive semidefinite matrix $Y \succeq 0$;

Exponential Cone

PICOS can also reformulate several constraints using the *exponential cone* `ExponentialCone`, as it is the case for example for `KullbackLeiblerConstraint`. PICOS provides dual values for `ExpConeConstraint`, as computed by the solver, but dualization of those constraints is not yet supported.

5.4 Numeric Tolerances

PICOS allows you to fine-tune how accurate your solution needs to be. Tolerances fall in three categories:

- **Feasibility** tolerances, abbreviated `fsb`, control the magnitude of constraint violation that is tolerated. The *integrality tolerance* also falls into this category.
- **Optimality** tolerances, abbreviated `opt`, control the maximum allowed deviation from the mathematically exact optimum solution and serve as a termination criterion. An exception is the the Simplex algorithm that uses the *dual feasibility* as its stopping criterion.
- The remaining tolerances are used at intermediate steps of specific algorithms, such as the *Markowitz threshold* used in a pivoting strategy of the Simplex algoritm.

Solvers differ in how they measure deviations from the ideal values. Some bound **absolute** values while others consider the deviation **in relation** to the magnitude of the numbers that occur in the problem. PICOS abbreviates the former measurement with `abs` and the latter with `rel`. If both measurements are supported by a solver, then the standard approach is to allow values if they are sufficiently accurate according to either one.

If solvers use a single value for **primal** and **dual** feasibility but PICOS is configured to use differing accuracies, supplied in the options with the `prim` and `dual` abbreviations respectively, it will supply the smaller of both values to such solvers.

By default, PICOS overrides the solver's default accuracies with common values, so that the choice of solver becomes transparent to you. Given that `P` is your problem instance, you can make PICOS respect the solvers' individual choices as follows:

```
>>> import picos
>>> P = picos.Problem()
>>> P.options["*_tol"] = None
```

5.4.1 Comparison Table

The table shows what tolerance *`options`* are supported by PICOS and each solver, and what their respective default value is.

Option	PICOS	CPLEX	CVXOPT	ECOS	GLPK	Gurobi	MOSEK	QICS	SCIP	SMCP
abs_prim_fsb_tol	10^{-8}	SX: 10^{-6}	unused	unused ?	SX: 10^{-7}	10^{-6} ?	SX: 10^{-6} LP: 10^{-8} ? CP: 10^{-8} ? QP: 10^{-8} ? NL: 10^{-8} ? IP: 10^{-6} ?	unused	SX: 10^{-6}	unused
rel_prim_fsb_tol	10^{-8}	LQ: 10^{-8} QC: 10^{-8}	CP: 10^{-7} NL: 10^{-7}	10^{-8} ?	unused ?	unused ?		10^{-8}	10^{-6}	10^{-8}
abs_dual_fsb_tol	10^{-8}	SX: 10^{-6}	unused	unused ?	SX: 10^{-7}	10^{-6}	SX: 10^{-6} LP: 10^{-8} ? CP: 10^{-8} ? QP: 10^{-8} ? NL: 10^{-8} ?	unused	SX: 10^{-7}	unused
rel_dual_fsb_tol	10^{-8}	LQ: 10^{-8} QC: 10^{-8}	CP: 10^{-7} NL: 10^{-7}	10^{-8} ?	unused ?	unused	SX: 10^{-12}	10^{-8}	10^{-6}	10^{-8}
abs_ipm_opt_tol	10^{-8}	unused	CP: 10^{-7} NL: 10^{-7}	10^{-8}	unused	unused	unused	unused	0	10^{-6}
rel_ipm_opt_tol	10^{-8}	LQ: 10^{-8} QC: 10^{-8}	CP: 10^{-6} NL: 10^{-6}	10^{-8}	unused	CO: 10^{-8} QC: 10^{-6}	LP: 10^{-8} CP: 10^{-7} QP: 10^{-7} NL: 10^{-6}	10^{-8}	0	10^{-6}
abs_bnb_opt_tol	10^{-6}	10^{-6}	no IP	10^{-6}	unused	10^{-10}	0	no IP	0	no IP
rel_bnb_opt_tol	10^{-4}	10^{-4}	no IP	10^{-3}	0	10^{-4}	10^{-4}	no IP	0	no IP
integrality_tol	10^{-5}	10^{-5}	no IP	10^{-4}	10^{-5}	10^{-5}	10^{-5}	no IP	unused	no IP
markowitz_tol	None	0.01	no SX	no SX	0.1	2^{-7}	unused ?	no SX	unused	no SX

Pooled options

- ECOS, CVXOPT, QICS, SCIP and SMCP merge *rel_prim_fsb_tol* and *rel_dual_fsb_tol*.
- CPLEX merges *rel_prim_fsb_tol*, *rel_dual_fsb_tol* and *rel_ipm_opt_tol*.
- SCIP appears to merge *abs_ipm_opt_tol* with *abs_bnb_opt_tol* and *rel_ipm_opt_tol* with *rel_bnb_opt_tol* with its `limits/absgap` and `limits/gap` options, respectively.

Legend

?	It is unclear whether an absolute or relative measure is used, or if an option is not available.
SX	Linear Programs via Simplex
LP	Linear Programs via Interior-Point Method
CP	Conic Programs
LQ	Linear and Quadratic Programs
QP	Quadratic Programs
QC	Quadratically Constrained (Quadratic) Programs
NL	Nonlinear Programs
IP	(Mixed) Integer Programs

Warning: This part of the documentation has not been touched for a while. It might be incomplete, reference deprecated functions or make a claim that does not apply to the latest version of PICOS any more. On the bright side, code listings are validated and still work. Watch your step!

5.5 Cheat Sheet

5.5.1 Manipulate expressions

Operator	Interpretation
`+`	addition
`+=`	inplace addition
`-`	substraction
`*`	multiplication
`^`	Hadamard (elementwise) product
`@`	Kronecker 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

5.5.2 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 $\ni$

5.5.3 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
partial_transpose	partial transposition
partial_trace	partial trace

5.5.4 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
quantrelentr	quantum relative entropy
quantkeydist	quantum key distribution

5.5.5 Create concave expressions

function	short doc
quantentr	quantum (von Neumann) entropy
quantcondentr	quantum conditional entropy

5.5.6 Create operator convex/concave expressions

function	short doc
oprelentr	operator relative entropy
mtxgeomean	(weighted) matrix geometric mean

5.5.7 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 $\{x : \|x\|_\infty \leq 1; \|x\|_1 \leq a\}$

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
write_to_file()	writes problem to a file

CHAPTER

SIX

API REFERENCE

PICOS is organized in a number of submodules and subpackages, 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. A notable exception are the tools for handling uncertain data that are found in the *picos.uncertain* namespace.

Modules

6.1 picos

A Python Interface to Conic Optimization Solvers.

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.

Exceptions

exception picos.**NotValued**

See *picos.valuable.NotValued*.

exception picos.**SolutionFailure**

See *picos.modeling.problem.SolutionFailure*.

Classes

class picos.**Ball**

See *picos.expressions.set_ball.Ball*.

class picos.**BinaryVariable**

See *picos.expressions.variables.BinaryVariable*.

class picos.**ComplexVariable**

See *picos.expressions.variables.ComplexVariable*.

class picos.**DetRootN**

See *picos.expressions.exp_detrootn.DetRootN*.

class picos.**Ellipsoid**

See *picos.expressions.set_ellipsoid.Ellipsoid*.

class picos.**Entropy**

See *picos.expressions.exp_entropy.Entropy*.

class picos.**ExponentialCone**

See *picos.expressions.cone_expcone.ExponentialCone*.

`class picos.FlowConstraint`
See *picos.constraints.con_flow.FlowConstraint*.

`class picos.GeometricMean`
See *picos.expressions.exp_geomean.GeometricMean*.

`class picos.HermitianVariable`
See *picos.expressions.variables.HermitianVariable*.

`class picos.IntegerVariable`
See *picos.expressions.variables.IntegerVariable*.

`class picos.LogSumExp`
See *picos.expressions.exp_logsumexp.LogSumExp*.

`class picos.Logarithm`
See *picos.expressions.exp_logarithm.Logarithm*.

`class picos.LowerTriangularVariable`
See *picos.expressions.variables.LowerTriangularVariable*.

`class picos.NegativeEntropy`
See *picos.expressions.exp_entropy.NegativeEntropy*.

`class picos.NonnegativeOrthant`
See *picos.expressions.cone_nno.NonnegativeOrthant*.

`class picos.Norm`
See *picos.expressions.exp_norm.Norm*.

`class picos.NuclearNorm`
See *picos.expressions.exp_nucnorm.NuclearNorm*.

`class picos.Objective`
See *picos.modeling.objective.Objective*.

`class picos.Options`
See *picos.modeling.options.Options*.

`class picos.PositiveSemidefiniteCone`
See *picos.expressions.cone_psd.PositiveSemidefiniteCone*.

`class picos.PowerTrace`
See *picos.expressions.exp_powtrace.PowerTrace*.

`class picos.Problem`
See *picos.modeling.problem.Problem*.

`class picos.ProductCone`
See *picos.expressions.cone_product.ProductCone*.

`class picos.RealVariable`
See *picos.expressions.variables.RealVariable*.

`class picos.RotatedSecondOrderCone`
See *picos.expressions.cone_rsoc.RotatedSecondOrderCone*.

`class picos.Samples`
See *picos.expressions.samples.Samples*.

`class picos.SecondOrderCone`
See *picos.expressions.cone_soc.SecondOrderCone*.

class picos.**Simplex**

See *picos.expressions.set_simplex.Simplex*.

class picos.**SkewSymmetricVariable**

See *picos.expressions.variables.SkewSymmetricVariable*.

class picos.**Solution**

See *picos.modeling.solution.Solution*.

class picos.**SpectralNorm**

See *picos.expressions.exp_specnorm.SpectralNorm*.

class picos.**SquaredNorm**

See *picos.expressions.exp_sqnorm.SquaredNorm*.

class picos.**SumExponentials**

See *picos.expressions.exp_sumexp.SumExponentials*.

class picos.**SumExtremes**

See *picos.expressions.exp_sumxtr.SumExtremes*.

class picos.**SymmetricVariable**

See *picos.expressions.variables.SymmetricVariable*.

class picos.**TheField**

See *picos.expressions.cone_trivial.TheField*.

class picos.**UpperTriangularVariable**

See *picos.expressions.variables.UpperTriangularVariable*.

class picos.**ZeroSpace**

See *picos.expressions.cone_trivial.ZeroSpace*.

Functions

picos.**Constant**()

See *picos.expressions.exp_affine.Constant*.

picos.**ascii**()

See *picos.glyphs.ascii*.

picos.**available_solvers**()

See *picos.solvers.available_solvers*.

picos.**ball**()

See *picos.expressions.algebra.ball*.

picos.**block**()

See *picos.expressions.algebra.block*.

picos.**default_charset**()

See *picos.glyphs.unicode*.

picos.**detrootn**()

See *picos.expressions.algebra.detrootn*.

picos.**diag**()

See *picos.expressions.algebra.diag*.

picos.**diag_vect**()

See *picos.expressions.algebra.diag_vect*.

picos.**exp**()

See *picos.expressions.algebra.exp*.

picos.**expcone**()

See *picos.expressions.algebra.expcone*.

picos.**find_assignment**()

See *picos.modeling.quicksolve.find_assignment*.

picos.**flow_Constraint**()

See *picos.expressions.algebra.flow_Constraint*.

picos.**geomean**()

See *picos.expressions.algebra.geomean*.

picos.**import_cbf**()

See *picos.modeling.file_in.import_cbf*.

picos.**kldiv**()

See *picos.expressions.algebra.kldiv*.

picos.**kron**()

See *picos.expressions.algebra.kron*.

picos.**kullback_leibler**()

See *picos.expressions.algebra.kullback_leibler*.

picos.**lambda_max**()

See *picos.expressions.algebra.lambda_max*.

picos.**lambda_min**()

See *picos.expressions.algebra.lambda_min*.

picos.**latin1**()

See *picos.glyphs.latin1*.

picos.**log**()

See *picos.expressions.algebra.log*.

picos.**logsumexp**()

See *picos.expressions.algebra.logsumexp*.

picos.**lse**()

See *picos.expressions.algebra.lse*.

picos.**maindiag**()

See *picos.expressions.algebra.maindiag*.

picos.**max**()

See *picos.expressions.algebra.max*.

picos.**maximize**()

See *picos.modeling.quicksolve.maximize*.

picos.**min**()

See *picos.expressions.algebra.min*.

picos.**minimize**()

See *picos.modeling.quicksolve.minimize*.

picos.**mtxgeomean**()

See *picos.expressions.algebra.oprelentr*.

picos.**new_param**()
: See *picos.expressions.algebra.new_param*.

picos.**norm**()
: See *picos.expressions.algebra.norm*.

picos.**oprelentr**()
: See *picos.expressions.algebra.oprelentr*.

picos.**partial_trace**()
: See *picos.expressions.algebra.partial_trace*.

picos.**partial_transpose**()
: See *picos.expressions.algebra.partial_transpose*.

picos.**patch_scipy_array_priority**()
: See *picos.valuable.patch_scipy_array_priority*.

picos.**quantcondentr**()
: See *picos.expressions.algebra.quantcondentr*.

picos.**quantentr**()
: See *picos.expressions.algebra.quantentr*.

picos.**quantkeydist**()
: See *picos.expressions.algebra.quantkeydist*.

picos.**quantrelentr**()
: See *picos.expressions.algebra.quantrelentr*.

picos.**rsoc**()
: See *picos.expressions.algebra.rsoc*.

picos.**simplex**()
: See *picos.expressions.algebra.simplex*.

picos.**soc**()
: See *picos.expressions.algebra.soc*.

picos.**sum**()
: See *picos.expressions.algebra.sum*.

picos.**sum_k_largest**()
: See *picos.expressions.algebra.sum_k_largest*.

picos.**sum_k_largest_lambda**()
: See *picos.expressions.algebra.sum_k_largest_lambda*.

picos.**sum_k_smallest**()
: See *picos.expressions.algebra.sum_k_smallest*.

picos.**sum_k_smallest_lambda**()
: See *picos.expressions.algebra.sum_k_smallest_lambda*.

picos.**sumexp**()
: See *picos.expressions.algebra.sumexp*.

picos.**trace**()
: See *picos.expressions.algebra.trace*.

picos.**tracepow**()
: See *picos.expressions.algebra.tracepow*.

picos.**truncated_simplex**()

See *picos.expressions.algebra.truncated_simplex*.

picos.**unicode**()

See *picos.glyphs.unicode*.

picos.**value**()

See *picos.expressions.data.value*.

Objects

picos.**I**

Create an identity matrix.

Example

```
>>> from picos import I
>>> print(I(3))
[ 1.00e+00     0          0    ]
[    0      1.00e+00     0    ]
[    0         0      1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c9e0>
```

picos.**J**

Create a matrix of all ones.

Example

```
>>> from picos import J
>>> print(J(2, 3))
[ 1.00e+00  1.00e+00  1.00e+00]
[ 1.00e+00  1.00e+00  1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655ca90>
```

picos.**O**

Create a zero matrix.

Example

```
>>> from picos import O
>>> print(O(2, 3))
[0 0 0]
[0 0 0]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c930>
```

6.1.1 picos.apidoc

Functions to support the automatic API documentation.

The environment variable `PICOS_WARN_MISSING_DOCSTRINGS` can be set to warn about missing docstrings for top level objects of a module.

Functions

`picos.apidoc.`**`api_end`**`(`*startGlobals*, *endGlobals*`)`

Mark the end of a module's content.

Store the returned value in __all__.

`picos.apidoc.`**`api_start`**`(`*theGlobals*`)`

Mark the start of a module's content.

Store the returned value and pass it to *`api_end`*.

6.1.2 picos.caching

Caching helpers.

Classes

`class picos.caching.`**`cached_property`**`(`*fget=None*, *fset=None*, *fdel=None*, *doc=None*`)`

Bases: `property`

A read-only property whose result is cached.

`__init__``(`*fget=None*, *fset=None*, *fdel=None*, *doc=None*`)`

`deleter``(`*fdel*`)`

Descriptor to obtain a copy of the property with a different deleter.

`getter``(`*fget*`)`

Descriptor to obtain a copy of the property with a different getter.

`setter``(`*fset*`)`

Descriptor to obtain a copy of the property with a different setter.

`class picos.caching.`**`cached_selfinverse_property`**`(`*fget=None*, *fset=None*, *fdel=None*, *doc=None*`)`

Bases: *`cached_property`*

A read-only, self-inverse property whose result is cached.

Functions

`picos.caching.`**`borrow_cache`**`(`*target*, *source*, *names*`)`

Copy cached values from one object to another.

Parameters

- **`target`** – The object to populate the cache of.
- **`source`** – The object to take cached values from.
- **`names`** – Names of cached properties or functions to borrow.

`picos.caching.`**`cached_selfinverse_unary_operator`**(*operator*)

Make a self-inverse unary operator method cache its result.

This is supposed to be used for property-like special methods such as `__neg__` where *`cached_property`* can't be used.

> **Warning:** The result returned by the wrapped operator must be a fresh object as it will be modified.

`picos.caching.`**`cached_unary_operator`**(*operator*)

Make a unary operator method cache its result.

This is supposed to be used for property-like special methods such as `__neg__` where *`cached_property`* can't be used.

`picos.caching.`**`empty_cache`**(*obj*)

Clear all cached values of an object.

`picos.caching.`**`unlocked_cached_properties`**(*obj*)

Unlock the setters of cached instance attributes.

Normally, cached attributes are read-only properties. When the user first reads them, the cache is populated with the value returned to the user, and successive reads will return the cached value.

The user is allowed to empty the cache by using `del` on the variable, but they may not assign a value to it. This context allows the programmer to manually populate the cache by assigning a value to the property.

Example

```
>>> from picos.caching import cached_property, unlocked_cached_properties
>>> class A:
...     @cached_property
...     def p(self):
...         return 1
...
>>> a = A()
>>> try:
...     a.p = 2
... except AttributeError:
...     print("Not possible.")
...
Not possible.
>>> with unlocked_cached_properties(a):
...     a.p = 2  # Populate the cache of a.p.
...
>>> a.p
2
```

Objects

`picos.caching.`**`CACHED_PREFIX`**

The prefix used for storing cached values.

Default value

```
'_cached_'
```

`picos.caching.`**`CACHED_PROP_UNLOCKED_TOKEN`**

An attribute name whose presence unlocks the setter of cached properties.

Default value

```
'_CACHED_PROPERTIES_ARE_UNLOCKED'
```

6.1.3 picos.constraints

Optimization constraint types.

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

`class picos.constraints.AbsoluteValueConstraint`

See `picos.constraints.con_absolute.AbsoluteValueConstraint`.

`class picos.constraints.AffineConstraint`

See `picos.constraints.con_affine.AffineConstraint`.

`class picos.constraints.BallUncertainNormConstraint`

See `picos.constraints.uncertain.ucon_ball_norm.BallUncertainNormConstraint`.

`class picos.constraints.ComplexAffineConstraint`

See `picos.constraints.con_affine.ComplexAffineConstraint`.

`class picos.constraints.ComplexLMIConstraint`

See `picos.constraints.con_lmi.ComplexLMIConstraint`.

`class picos.constraints.ComplexMatrixGeoMeanEpiConstraint`

See `picos.constraints.con_mtxgeomean.ComplexMatrixGeoMeanEpiConstraint`.

`class picos.constraints.ComplexMatrixGeoMeanHypoConstraint`

See `picos.constraints.con_mtxgeomean.ComplexMatrixGeoMeanHypoConstraint`.

`class picos.constraints.ComplexOpRelEntropyConstraint`

See `picos.constraints.con_oprelentr.ComplexOpRelEntropyConstraint`.

`class picos.constraints.ComplexQuantCondEntropyConstraint`

See `picos.constraints.con_quantcondentr.ComplexQuantCondEntropyConstraint`.

`class picos.constraints.ComplexQuantKeyDistributionConstraint`

See `picos.constraints.con_quantkeydist.ComplexQuantKeyDistributionConstraint`.

`class picos.constraints.ComplexQuantRelEntropyConstraint`

See `picos.constraints.con_quantrelentr.ComplexQuantRelEntropyConstraint`.

`class picos.constraints.ComplexTrMatrixGeoMeanEpiConstraint`

See `picos.constraints.con_mtxgeomean.ComplexTrMatrixGeoMeanEpiConstraint`.

`class picos.constraints.ComplexTrMatrixGeoMeanHypoConstraint`

See `picos.constraints.con_mtxgeomean.ComplexTrMatrixGeoMeanHypoConstraint`.

`class picos.constraints.ComplexTrOpRelEntropyConstraint`

See `picos.constraints.con_oprelentr.ComplexTrOpRelEntropyConstraint`.

`class picos.constraints.ConicConstraint`

See `picos.constraints.constraint.ConicConstraint`.

class picos.constraints.**ConicQuadraticConstraint**

See *picos.constraints.con_quadratic.ConicQuadraticConstraint*.

class picos.constraints.**ConicallyUncertainAffineConstraint**

See *picos.constraints.uncertain.ucon_conic_aff.ConicallyUncertainAffineConstraint*.

class picos.constraints.**Constraint**

See *picos.constraints.constraint.Constraint*.

class picos.constraints.**ConstraintConversion**

See *picos.constraints.constraint.ConstraintConversion*.

class picos.constraints.**ConstraintType**

See *picos.constraints.constraint.ConstraintType*.

class picos.constraints.**ConvexQuadraticConstraint**

See *picos.constraints.con_quadratic.ConvexQuadraticConstraint*.

class picos.constraints.**DetRootNConstraint**

See *picos.constraints.con_detrootn.DetRootNConstraint*.

class picos.constraints.**DummyConstraint**

See *picos.constraints.con_dummy.DummyConstraint*.

class picos.constraints.**ExpConeConstraint**

See *picos.constraints.con_expcone.ExpConeConstraint*.

class picos.constraints.**ExtremumConstraint**

See *picos.constraints.con_extremum.ExtremumConstraint*.

class picos.constraints.**FlowConstraint**

See *picos.constraints.con_flow.FlowConstraint*.

class picos.constraints.**GeometricMeanConstraint**

See *picos.constraints.con_geomean.GeometricMeanConstraint*.

class picos.constraints.**KullbackLeiblerConstraint**

See *picos.constraints.con_kldiv.KullbackLeiblerConstraint*.

class picos.constraints.**LMIConstraint**

See *picos.constraints.con_lmi.LMIConstraint*.

class picos.constraints.**LogConstraint**

See *picos.constraints.con_log.LogConstraint*.

class picos.constraints.**LogSumExpConstraint**

See *picos.constraints.con_logsumexp.LogSumExpConstraint*.

class picos.constraints.**MatrixGeoMeanEpiConstraint**

See *picos.constraints.con_mtxgeomean.MatrixGeoMeanEpiConstraint*.

class picos.constraints.**MatrixGeoMeanHypoConstraint**

See *picos.constraints.con_mtxgeomean.MatrixGeoMeanHypoConstraint*.

class picos.constraints.**MatrixNormConstraint**

See *picos.constraints.con_matnorm.MatrixNormConstraint*.

class picos.constraints.**MomentAmbiguousExtremumAffineConstraint**

See *picos.constraints.uncertain.ucon_mom_pwl.MomentAmbiguousExtremumAffineConstraint*.

class picos.constraints.**MomentAmbiguousSquaredNormConstraint**

See *picos.constraints.uncertain.ucon_mom_sqnorm.MomentAmbiguousSquaredNormConstraint*.

class picos.constraints.**NonconvexQuadraticConstraint**

See *picos.constraints.con_quadratic.NonconvexQuadraticConstraint*.

class picos.constraints.**NuclearNormConstraint**

See *picos.constraints.con_matnorm.NuclearNormConstraint*.

class picos.constraints.**OpRelEntropyConstraint**

See *picos.constraints.con_oprelentr.OpRelEntropyConstraint*.

class picos.constraints.**PowerTraceConstraint**

See *picos.constraints.con_powtrace.PowerTraceConstraint*.

class picos.constraints.**ProductConeConstraint**

See *picos.constraints.con_prodcone.ProductConeConstraint*.

class picos.constraints.**QuantCondEntropyConstraint**

See *picos.constraints.con_quantcondentr.QuantCondEntropyConstraint*.

class picos.constraints.**QuantKeyDistributionConstraint**

See *picos.constraints.con_quantkeydist.QuantKeyDistributionConstraint*.

class picos.constraints.**QuantRelEntropyConstraint**

See *picos.constraints.con_quantrelentr.QuantRelEntropyConstraint*.

class picos.constraints.**RSOCConstraint**

See *picos.constraints.con_rsoc.RSOCConstraint*.

class picos.constraints.**SOCConstraint**

See *picos.constraints.con_soc.SOCConstraint*.

class picos.constraints.**ScenarioUncertainConicConstraint**

See *picos.constraints.uncertain.ucon_scen_conic.ScenarioUncertainConicConstraint*.

class picos.constraints.**SimplexConstraint**

See *picos.constraints.con_simplex.SimplexConstraint*.

class picos.constraints.**SpectralNormConstraint**

See *picos.constraints.con_matnorm.SpectralNormConstraint*.

class picos.constraints.**SquaredNormConstraint**

See *picos.constraints.con_sqnorm.SquaredNormConstraint*.

class picos.constraints.**SumExponentialsConstraint**

See *picos.constraints.con_sumexp.SumExponentialsConstraint*.

class picos.constraints.**SumExtremesConstraint**

See *picos.constraints.con_sumxtr.SumExtremesConstraint*.

class picos.constraints.**TrMatrixGeoMeanEpiConstraint**

See *picos.constraints.con_mtxgeomean.TrMatrixGeoMeanEpiConstraint*.

class picos.constraints.**TrMatrixGeoMeanHypoConstraint**

See *picos.constraints.con_mtxgeomean.TrMatrixGeoMeanHypoConstraint*.

class picos.constraints.**TrOpRelEntropyConstraint**

See *picos.constraints.con_oprelentr.TrOpRelEntropyConstraint*.

class picos.constraints.**VectorNormConstraint**

See *picos.constraints.con_vecnorm.VectorNormConstraint*.

class picos.constraints.**WassersteinAmbiguousExtremumAffineConstraint**

See *picos.constraints.uncertain.ucon_ws_pwl.WassersteinAmbiguousExtremumAffineConstraint*.

class picos.constraints.**WassersteinAmbiguousSquaredNormConstraint**

See *picos.constraints.uncertain.ucon_ws_sqnorm.WassersteinAmbiguousSquaredNormConstraint*.

class picos.constraints.**WeightedSumConstraint**

See *picos.constraints.con_wsum.WeightedSumConstraint*.

picos.constraints.con_absolute

Implementation of *AbsoluteValueConstraint*.

Classes

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

Bases: *Constraint*

Upper bound on an absolute value.

class AffineConversion

Bases: *ConstraintConversion*

Upper bound on an absolute value to affine inequality conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod dual(*auxVarPrimals*, *auxConDuals*, *options*)

Implement *dual*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*signedScalar*, *upperBound*)

Construct an *AbsoluteValueConstraint*.

Parameters

- **signedScalar** (AffineExpression) – A scalar expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

picos.constraints.con_affine

Affine constraint types.

Classes

class picos.constraints.con_affine.**AffineConstraint**(*lhs*, *relation*, *rhs*, *customString=None*)

Bases: *ConicConstraint*

An equality or inequality between two affine expressions.

__init__(*lhs*, *relation*, *rhs*, *customString=None*)

Construct an *AffineConstraint*.

Parameters

- **lhs** (AffineExpression) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

- `customString` (*str*) – Optional string description.

`bounded_linear_form()`

Bounded linear form of the constraint.

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.

`property conic_membership_form`

Implement for `ConicConstraint`.

`property ge0`

Expression constrained to be greater than or equal to zero.

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

`property greater`

Greater-or-equal side of the constraint.

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

`property le0`

Expression constrained to be lower than or equal to zero.

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

`property lmr`

Left hand side minus right hand side.

`property nnVar`

The variable posed nonnegative, or `None`.

`property rml`

Right hand side minus left hand side.

`property smaller`

Smaller-or-equal side of the constraint.

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

`class picos.constraints.con_affine.ComplexAffineConstraint`(*lhs*, *rhs*, *customString=None*)

Bases: `ConicConstraint`

An equality between affine expressions, at least one being complex.

`class RealConversion`

Bases: `ConstraintConversion`

Complex affine equality to real affine equality conversion.

`classmethod convert`(*con*, *options*)

Implement `convert`.

`classmethod dual`(*auxVarPrimals*, *auxConDuals*, *options*)

Implement `dual`.

`classmethod predict`(*subtype*, *options*)

Implement `predict`.

`__init__`(*lhs*, *rhs*, *customString=None*)

Construct a *`ComplexAffineConstraint`*.

Parameters

- **`lhs`** (`AffineExpression`) – Left hand side expression.
- **`rhs`** (`AffineExpression`) – Right hand side expression.
- **`customString`** (*`str`*) – Optional string description.

`property conic_membership_form`

Implement for *`ConicConstraint`*.

picos.constraints.con_detrootn

Implementation of *`DetRootNConstraint`*.

Classes

`class picos.constraints.con_detrootn.`**`DetRootNConstraint`**(*detRootN*, *lowerBound*)

Bases: *`Constraint`*

Lower bound on the n-th root of a matrix determinant.

`class Conversion`

Bases: *`ConstraintConversion`*

n-th root of a matrix determinant constraint conversion.

`classmethod convert`(*con*, *options*)

Implement *`convert`*.

`classmethod predict`(*subtype*, *options*)

Implement *`predict`*.

`__init__`(*detRootN*, *lowerBound*)

Construct a *`DetRootNConstraint`*.

Parameters

- **`detRootN`** (`DetRootN`) – Constrained expression.
- **`lowerBound`** (`AffineExpression`) – Lower bound on the expression.

picos.constraints.con_dummy

Implementation of *`DummyConstraint`*.

Classes

`class picos.constraints.con_dummy.`**`DummyConstraint`**(*x*)

Bases: *`ConicConstraint`*

An explicit way to *not* put a bound on an affine expression.

This is produced when declaring an expression a member of the trivial *`TheField`* cone.

A constraint of this type can be used to pass a variable to a solver that does not otherwise appear in the problem.

`__init__`(*x*)

Construct a `DummyConstraint`.

`property conic_membership_form`

Implement for `ConicConstraint`.

picos.constraints.con_expcone

Implementation of `ExpConeConstraint`.

Classes

`class picos.constraints.con_expcone.`**`ExpConeConstraint`**(*element*, *customString=None*)

Bases: `Constraint`

Exponential cone membership constraint.

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

`__init__`(*element*, *customString=None*)

Construct an `ExpConeConstraint`.

Parameters

- **`element`** (`AffineExpression`) – Three-dimensional expression representing the vector (x, y, z).
- **`customString`** (`str`) – Optional string description.

`property conic_membership_form`

Implement for `ConicConstraint`.

`property x`

`property y`

`property z`

picos.constraints.con_extremum

Implementation of `ExtremumConstraint`.

Classes

`class picos.constraints.con_extremum.`**`ExtremumConstraint`**(*extremum*, *relation*, *rhs*)

Bases: `Constraint`

Bound on a maximum (minimum) over convex (concave) functions.

`class Conversion`

Bases: `ConstraintConversion`

Bound on a maximum or minimum of functions conversion.

`classmethod convert`(*con*, *options*)

Implement `convert`.

`classmethod predict`(*subtype*, *options*)

Implement `predict`.

`__init__`(*extremum*, *relation*, *rhs*)

Construct a *`ExtremumConstraint`*.

Parameters

- **`extremum`** (`Extremum`) – Left hand side expression.
- **`relation`** (*`str`*) – Constraint relation symbol.
- **`rhs`** (`AffineExpression`) – Right hand side expression.

picos.constraints.con_flow

Implementation of *`FlowConstraint`*.

Classes

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

Bases: *`Constraint`*

Network flow constraint.

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

`class` **`Conversion`**

Bases: *`ConstraintConversion`*

Network flow constraint conversion.

`classmethod` **`convert`**(*con*, *options*)

Implement *`convert`*.

`classmethod` **`predict`**(*subtype*, *options*)

Implement *`predict`*.

`__init__`(*G*, *f*, *source*, *sink*, *flow_value*, *capacity=None*, *graphName=''*)

Construct 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 *`AffineExpression`*.
- **`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.

draw()

Draw the graph.

replace_mutables(*mapping*)

Make the constraint concern a different set of mutables.

See *replace_mutables* for more.

property mutables

All mutables referenced by the constraint.

picos.constraints.con_geomean

Implementation of *GeometricMeanConstraint*.

Classes

class picos.constraints.con_geomean.**GeometricMeanConstraint**(*geoMean*, *lowerBound*)

Bases: *Constraint*

Lower bound on a geometric mean.

class RSOCConversion

Bases: *ConstraintConversion*

Geometric mean to rotated second order cone constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*geoMean*, *lowerBound*)

Construct a *GeometricMeanConstraint*.

Parameters

- **lhs** (GeometricMean) – Constrained expression.
- **lowerBound** (AffineExpression) – Lower bound on the expression.

picos.constraints.con_kldiv

Implementation of *KullbackLeiblerConstraint*.

Classes

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

Bases: *Constraint*

Upper bound on a Kullback-Leibler divergence.

This is the upper bound on a negative or relative entropy, both represented by *NegativeEntropy*.

class ExpConeConversion

Bases: *ConstraintConversion*

Kullback-Leibler to exponential cone constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*divergence*, *upperBound*)

Construct a *KullbackLeiblerConstraint*.

Parameters

- **divergence** (NegativeEntropy) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property denominator

The y of the divergence, or 1.

property numerator

The x of the divergence.

picos.constraints.con_lmi

Linear matrix inequalities.

Classes

class picos.constraints.con_lmi.**ComplexLMIConstraint**(*lhs*, *relation*, *rhs*, *customString=None*)

Bases: *LMIConstraint*

Complex linear matrix inequality.

class RealConversion

Bases: *ConstraintConversion*

Complex LMI to real LMI conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod dual(*auxVarPrimals*, *auxConDuals*, *options*)

Implement *dual*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

class picos.constraints.con_lmi.**LMIConstraint**(*lhs*, *relation*, *rhs*, *customString=None*)

Bases: *ConicConstraint*

Linear matrix inequality.

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

__init__(*lhs*, *relation*, *rhs*, *customString=None*)

Construct a *LMIConstraint*.

Parameters

- **lhs** (AffineExpression) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

- **customString** (*str*) – Optional string description.

property conic_membership_form

Implement for *ConicConstraint*.

property greater

The greater-or-equal side expression.

property nnd

The matrix expression posed to be positive semidefinite.

property npd

The matrix expression posed to be negative semidefinite.

property nsd

The matrix expression posed to be negative semidefinite.

property psd

The matrix expression posed to be positive semidefinite.

property semidefVar

The variable posed positive semidefinite, or None.

property smaller

The smaller-or-equal side expression.

picos.constraints.con_log

Implementation of *LogConstraint*.

Classes

class picos.constraints.con_log.**LogConstraint**(*log*, *lowerBound*)

Bases: *Constraint*

Lower bound on a logarithm.

class ExpConeConversion

Bases: *ConstraintConversion*

Bound on a logarithm to exponential cone constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*log*, *lowerBound*)

Construct a *LogConstraint*.

Parameters

- **log** (Logarithm) – Constrained expression.
- **lowerBound** (AffineExpression) – Lower bound on the expression.

picos.constraints.con_logsumexp

Implementation of *LogSumExpConstraint*.

Classes

class picos.constraints.con_logsumexp.**LogSumExpConstraint**(*lse*, *upperBound*)

Bases: *Constraint*

Upper bound on a logarithm of a sum of exponentials.

class ExpConeConversion

Bases: *ConstraintConversion*

Bound on a log-sum-exp to exponential cone constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod dual(*auxVarPrimals*, *auxConDuals*, *options*)

Implement *dual*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*lse*, *upperBound*)

Construct a *LogSumExpConstraint*.

Parameters

- **lse** (LogSumExp) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property exponents

The affine exponents of the bounded log-sum-exp expression.

property le0

The *LogSumExp* posed to be at most zero.

picos.constraints.con_matnorm

Implementation of matrix norm constraints.

Classes

class picos.constraints.con_matnorm.**MatrixNormConstraint**(*norm*, *upperBound*)

Bases: *Constraint*

Upper bound on a matrix (p, q)-norm.

class VectorNormConversion

Bases: *ConstraintConversion*

Upper bound on a (p, q)-norm constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

`__init__`(*norm*, *upperBound*)

Construct a *`MatrixNormConstraint`*.

Parameters

- **`norm`** (`Norm`) – The norm.
- **`upperBound`** (`AffineExpression`) – The scalar upper bound.

`class picos.constraints.con_matnorm.`**`NuclearNormConstraint`**(*norm*, *upperBound*)

Bases: *`Constraint`*

Nuclear norm of a matrix.

`class Conversion`

Bases: *`ConstraintConversion`*

Nuclear norm constraint conversion.

`classmethod convert`(*con*, *options*)

Implement *`convert`*.

`classmethod predict`(*subtype*, *options*)

Implement *`predict`*.

`__init__`(*norm*, *upperBound*)

Construct a *`NuclearNormConstraint`*.

Parameters

- **`norm`** (`NuclearNorm`) – Constrained nuclear norm
- **`upperBound`** (`AffineExpression`) – Upper bound on the expression.

`class picos.constraints.con_matnorm.`**`SpectralNormConstraint`**(*norm*, *upperBound*)

Bases: *`Constraint`*

Spectral norm of a matrix.

`class Conversion`

Bases: *`ConstraintConversion`*

Spectral norm constraint conversion.

`classmethod convert`(*con*, *options*)

Implement *`convert`*.

`classmethod predict`(*subtype*, *options*)

Implement *`predict`*.

`__init__`(*norm*, *upperBound*)

Construct a *`SpectralNormConstraint`*.

Parameters

- **`norm`** (`SpectralNorm`) – Constrained spectral norm
- **`upperBound`** (`AffineExpression`) – Upper bound on the expression.

picos.constraints.con_mtxgeomean

Matrix geometric mean constraints.

Classes

class picos.constraints.con_mtxgeomean.**ComplexMatrixGeoMeanEpiConstraint**(*divergence*, *upperBound*)

Bases: *MatrixGeoMeanEpiConstraint*

Epigraph of a complex convex matrix geometric mean.

class picos.constraints.con_mtxgeomean.**ComplexMatrixGeoMeanHypoConstraint**(*divergence*, *lowerBound*)

Bases: *MatrixGeoMeanHypoConstraint*

Hypograph of a complex concave matrix geometric mean.

class picos.constraints.con_mtxgeomean.**ComplexTrMatrixGeoMeanEpiConstraint**(*divergence*, *upperBound*)

Bases: *TrMatrixGeoMeanEpiConstraint*

Upper bound of trace of a complex convex operator relative entropy.

class picos.constraints.con_mtxgeomean.**ComplexTrMatrixGeoMeanHypoConstraint**(*divergence*, *lowerBound*)

Bases: *TrMatrixGeoMeanHypoConstraint*

Lower bound of trace of a complex concave operator relative entropy.

class picos.constraints.con_mtxgeomean.**MatrixGeoMeanEpiConstraint**(*divergence*, *upperBound*)

Bases: *Constraint*

Epigraph of a convex matrix geometric mean.

This is the upper bound, in the Loewner order, of a convex matrix geometric mean, represented by *MatrixGeometricMean*.

__init__(*divergence*, *upperBound*)

Construct a *MatrixGeoMeanEpiConstraint*.

Parameters

- **divergence** (MatrixGeometricMean) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

property power

The power p.

class picos.constraints.con_mtxgeomean.**MatrixGeoMeanHypoConstraint**(*divergence*, *lowerBound*)

Bases: *Constraint*

Hypograph of a concave matrix geometric mean.

This is the lower bound, in the Loewner order, of a concave matrix geometric mean, represented by *MatrixGeometricMean*.

`__init__`(*divergence*, *lowerBound*)

Construct a *`MatrixGeoMeanEpiConstraint`*.

Parameters

- **`divergence`** (`MatrixGeometricMean`) – Constrained expression.
- **`lowerBound`** (`AffineExpression`) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

property power

The power p.

`class picos.constraints.con_mtxgeomean.`**`TrMatrixGeoMeanEpiConstraint`**(*divergence*, *upperBound*)

Bases: *`Constraint`*

Upper bound of trace of a convex operator relative entropy.

This is the upper bound on the trace of a convex matrix geometric mean, represented by *`TrMatrixGeometricMean`*.

`__init__`(*divergence*, *upperBound*)

Construct a *`MatrixGeoMeanEpiConstraint`*.

Parameters

- **`divergence`** (`TrMatrixGeometricMean`) – Constrained expression.
- **`upperBound`** (`AffineExpression`) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

property power

The power p.

`class picos.constraints.con_mtxgeomean.`**`TrMatrixGeoMeanHypoConstraint`**(*divergence*, *lowerBound*)

Bases: *`Constraint`*

Lower bound of trace of a concave operator relative entropy.

This is the lower bound on the trace of a concave matrix geometric mean, represented by *`TrMatrixGeometricMean`*.

`__init__`(*divergence*, *lowerBound*)

Construct a *`MatrixGeoMeanEpiConstraint`*.

Parameters

- **`divergence`** (`TrMatrixGeometricMean`) – Constrained expression.
- **`lowerBound`** (`AffineExpression`) – Lower bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

property power

The power p.

picos.constraints.con_oprelentr

Operator relative entropy constraints.

Classes

`class picos.constraints.con_oprelentr.`**ComplexOpRelEntropyConstraint**(*divergence*, *upperBound*)

Bases: *OpRelEntropyConstraint*

Epigraph of a complex operator relative entropy.

`class picos.constraints.con_oprelentr.`**ComplexTrOpRelEntropyConstraint**(*divergence*, *upperBound*)

Bases: *TrOpRelEntropyConstraint*

Upper bound of trace of complex operator relative entropy.

`class picos.constraints.con_oprelentr.`**OpRelEntropyConstraint**(*divergence*, *upperBound*)

Bases: *Constraint*

Epigraph of an operator relative entropy.

This is the upper bound, in the Loewner order, of an operator relative entropy, represented by *OperatorRelativeEntropy*.

__init__(*divergence*, *upperBound*)

Construct a *OpRelEntropyConstraint*.

Parameters

- **divergence** (OperatorRelativeEntropy) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

`class picos.constraints.con_oprelentr.`**TrOpRelEntropyConstraint**(*divergence*, *upperBound*)

Bases: *Constraint*

Upper bound of trace of operator relative entropy.

This is the upper bound on the trace of an operator relative entropy, represented by *TrOperatorRelativeEntropy*.

__init__(*divergence*, *upperBound*)

Construct a *TrOpRelEntropyConstraint*.

Parameters

- **divergence** (TrOperatorRelativeEntropy) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence.

picos.constraints.con_powtrace

Implementation of *PowerTraceConstraint*.

Classes

`class picos.constraints.con_powtrace.`**PowerTraceConstraint**(*power*, *relation*, *rhs*)

Bases: *Constraint*

Bound on the trace over the p-th power of a matrix.

For scalar expressions, this is simply a bound on their p-th power.

class Conversion

Bases: *ConstraintConversion*

Bound on the p-th power of a trace constraint conversion.

The conversion is based on this paper.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*power*, *relation*, *rhs*)

Construct a *PowerTraceConstraint*.

Parameters

- **ower** (PowerTrace) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

is_trace()

Whether the bound concerns a trace as opposed to a scalar.

property lhs

picos.constraints.con_prodcone

Implementation of *ProductConeConstraint*.

Classes

class picos.constraints.con_prodcone.**ProductConeConstraint**(*element*, *cone*)

Bases: *ConicConstraint*

Confines an element inside a real Cartesian product cone.

class Conversion

Bases: *ConstraintConversion*

Cartesian product cone membership conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod dual(*auxVarPrimals*, *auxConDuals*, *options*)

Implement *dual*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*element*, *cone*)

Construct a *ProductConeConstraint*.

Parameters

- **element** (AffineExpression) – The element confined in the product cone.
- **cone** (ProductCone) – The product cone.

property conic_membership_form

Implement for *ConicConstraint*.

picos.constraints.con_quadratic

Quadratic constraint types.

Classes

class picos.constraints.con_quadratic.**ConicQuadraticConstraint**(*lhs*, *relation*, *rhs*)

Bases: *NonconvexQuadraticConstraint*

Bound on a *nearly* convex quadratic expression.

Nearly convex means that the bilinear form representing the quadratic part of the expression that the constraint poses to be at most zero has exactly one negative eigenvalue. More precisely, if the constraint is of the form $x^TQx + a^Tx + b \leq x^TRx + b^Tx + c$, then *Q* - *R* needs to have exactly one negative eigenvalue for the constraint to be nearly convex. Such constraints can be posed as conic constraints under an additional assumption that some affine term is nonnegative.

At this point, this class may only be used for nonconvex constraints of the form $x^TQx + p^Tx + q \leq (a^Tx + b)(c^Tx + d)$ with $x^TQx + p^Tx + q$ representable as a squared norm. In this case, the additional assumptions required for a conic reformulation are $a^Tx + b \geq 0$ and $b^Tx + c \geq 0$.

Whether a constraint of this type is strengthened to a conic constraint or relabeled as a *NonconvexQuadraticConstraint* depends on the *assume_conic* option.

Example

```
>>> from picos import Options, RealVariable
>>> x, y, z = RealVariable("x"), RealVariable("y"), RealVariable("z")
>>> C = x**2 + 1 <= y*z; C
<Conic Quadratic Constraint: x² + 1 ≤ y·z>
>>> P = C.__class__.Conversion.convert(C, Options(assume_conic=True))
>>> list(P.constraints.values())[0]
<4×1 RSOC Constraint: ‖fullroot(x² + 1)‖² ≤ y·z ∨ y, z ≥ 0>
>>> Q = C.__class__.Conversion.convert(C, Options(assume_conic=False))
>>> list(Q.constraints.values())[0]
<Nonconvex Quadratic Constraint: x² + 1 ≤ y·z>
```

Note: Solver implementations must not support this constraint type so that the user's choice for the *assume_conic* option is respected.

class Conversion

Bases: `ConstraintConversion`

Nearly convex quadratic to (rotated) second order cone conversion.

classmethod convert(*con*, *options*)

Implement `convert`.

classmethod predict(*subtype*, *options*)

Implement `predict`.

__init__(*lhs*, *relation*, *rhs*)

Construct a `ConicQuadraticConstraint`.

See `NonconvexQuadraticConstraint.__init__` for more.

class picos.constraints.con_quadratic.**ConvexQuadraticConstraint**(*lhs*, *relation*, *rhs*)

Bases: `NonconvexQuadraticConstraint`

Bound on a convex quadratic expression.

class ConicConversion

Bases: `ConstraintConversion`

Convex quadratic to (rotated) second order cone conversion.

classmethod convert(*con*, *options*)

Implement `convert`.

classmethod predict(*subtype*, *options*)

Implement `predict`.

__init__(*lhs*, *relation*, *rhs*)

Construct a `ConvexQuadraticConstraint`.

See `NonconvexQuadraticConstraint.__init__` for more.

class picos.constraints.con_quadratic.**NonconvexQuadraticConstraint**(*lhs*, *relation*, *rhs*)

Bases: `Constraint`

Bound on a nonconvex quadratic expression.

__init__(*lhs*, *relation*, *rhs*)

Construct a `NonconvexQuadraticConstraint`.

Parameters

- **lhs** (`QuadraticExpression` *or* `AffineExpression`) – Left hand side quadratic or affine expression.

- **relation** (*str*) – Constraint relation symbol.
- **rhs** (QuadraticExpression *or* AffineExpression) – Right hand side quadratic or affine expression.

property greater

Greater-or-equal side of the constraint.

property le0

Quadratic expression constrained to be at most zero.

property smaller

Smaller-or-equal side of the constraint.

picos.constraints.con_quantcondentr

Implementation of *QuantCondEntropyConstraint*.

Classes

class picos.constraints.con_quantcondentr.ComplexQuantCondEntropyConstraint(*function*, *lowerBound*)

Bases: *QuantCondEntropyConstraint*

Lower bound on a complex quantum conditional entropy.

class picos.constraints.con_quantcondentr.QuantCondEntropyConstraint(*function*, *lowerBound*)

Bases: *Constraint*

Lower bound on a quantum conditional entropy.

This is the lower bound on a quantum conditional entropy, represented by *QuantumConditionalEntropy*.

__init__(*function*, *lowerBound*)

Construct a *QuantCondEntropyConstraint*.

Parameters

- **function** (QuantumConditionalEntropy) – Constrained expression.
- **lowerBound** (AffineExpression) – Lower bound on the expression.

property X

The X of the function.

property dimensions

The dimensions of the subsystems of X.

property subsystems

The subsystems being traced out of X.

picos.constraints.con_quantkeydist

Implementation of *QuantKeyDistributionConstraint*.

Classes

`class picos.constraints.con_quantkeydist.`**`ComplexQuantKeyDistributionConstraint`**`(`*function*, *upperBound*`)`

Bases: *QuantKeyDistributionConstraint*

Upper bound on a complex quantum key distribution function.

`class picos.constraints.con_quantkeydist.`**`QuantKeyDistributionConstraint`**`(`*function*, *upperBound*`)`

Bases: *Constraint*

Upper bound on a quantum key distribution function.

This is the upper bound on a quantum key distribution function, represented by *QuantumKeyDistribution*.

`__init__``(`*function*, *upperBound*`)`

Construct a *QuantKeyDistributionConstraint*.

Parameters

- **function** (QuantumKeyDistribution) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

`property K_list`

The Kraus operators K_i of $\mathcal{G}$.

`property X`

The X of the function.

`property Z_list`

The Kraus operators Z_i of $\mathcal{Z}$.

`property dimensions`

The dimensions of the subsystems of X.

`property subsystems`

The subsystems being block-diagonalized of X.

picos.constraints.con_quantrelentr

Implementation of *QuantRelEntropyConstraint*.

Classes

`class picos.constraints.con_quantrelentr.`**`ComplexQuantRelEntropyConstraint`**`(`*divergence*, *upperBound*`)`

Bases: *QuantRelEntropyConstraint*

Upper bound on a complex quantum relative entropy.

class picos.constraints.con_quantrelentr.**QuantRelEntropyConstraint**(*divergence*, *upperBound*)

Bases: *Constraint*

Upper bound on a quantum relative entropy.

This is the upper bound on a negative or relative quantum entropy, both represented by *NegativeQuantumEntropy*.

__init__(*divergence*, *upperBound*)

Construct a *QuantRelEntropyConstraint*.

Parameters

- **divergence** (NegativeQuantumEntropy) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property X

The X of the divergence.

property Y

The Y of the divergence, or $\mathbb{I}$.

picos.constraints.con_rsoc

Rotated second order cone constraints.

Classes

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

Bases: *ConicConstraint*

Rotated second order cone membership constraint.

__init__(*normedExpression*, *upperBoundFactor1*, *upperBoundFactor2=None*, *customString=None*)

Construct a *RSOCConstraint*.

Parameters

- **normedExpression** (AffineExpression) – Expression under the norm.
- **upperBoundFactor1** (AffineExpression) – First of the two scalar factors that make the upper bound on the normed expression.
- **upperBoundFactor2** (AffineExpression) – Second of the two scalar factors that make the upper bound on the normed expression.
- **customString** (*str*) – Optional string description.

property conic_membership_form

Implement for *ConicConstraint*.

picos.constraints.con_simplex

Implementation of *SimplexConstraint*.

Classes

class picos.constraints.con_simplex.**SimplexConstraint**(*simplex*, *element*)

Bases: *Constraint*

(Symmetrized, truncated) simplex membership constraint.

class AffineConversion

Bases: *ConstraintConversion*

Simplex membership to affine constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*simplex*, *element*)

Construct a *SimplexConstraint*.

Parameters

element (AffineExpression) – Expression in the simplex.

picos.constraints.con_soc

Second order conce constraints.

Classes

class picos.constraints.con_soc.**SOCConstraint**(*normedExpression*, *upperBound*, *customString=None*)

Bases: *ConicConstraint*

Second order (2-norm, Lorentz) cone membership constraint.

__init__(*normedExpression*, *upperBound*, *customString=None*)

Construct a *SOCConstraint*.

Parameters

- **normedExpression** (AffineExpression) – Expression under the norm.
- **upperBound** (AffineExpression) – Upper bound on the normed expression.
- **customString** (*str*) – Optional string description.

property conic_membership_form

Implement for *ConicConstraint*.

property unit_ball_form

The constraint in Euclidean norm unit ball membership form.

If this constraint has the form $\|E(X)\|_F \leq c$ with $c > 0$ constant and $E(X)$ an affine expression of a single (matrix) variable X with $y := \operatorname{vec}(E(X)) = A \operatorname{vec}(X) + b$ for some invertible matrix A and a vector b, then we have $\operatorname{vec}(X) = A^{-1}(y - b)$ and we can write the elementwise vectorization of the constraint's feasible region as

$$\begin{aligned}
&\{\operatorname{vec}(X) \mid \|E(X)\|_F \leq c\} \\
={}& \{\operatorname{vec}(X) \mid \|A \operatorname{vec}(X) + b\|_2 \leq c\} \\
={}& \left\{A^{-1}(y - b) \mid \|y\|_2 \leq c\right\} \\
={}& \left\{A^{-1}(y - b) \mid \|c^{-1}y\|_2 \leq 1\right\} \\
={}& \left\{A^{-1}(cy - b) \mid \|y\|_2 \leq 1\right\}.
\end{aligned}$$

Therefor we can repose the constraint as two constraints:

$$\|E(X)\|_F \leq c \iff \exists y : \begin{cases} \operatorname{vec}(X) = A^{-1}(cy - b) \\ \|y\|_2 \leq 1. \end{cases}$$

This method returns the quadruple $(X, A^{-1}(cy - b), y, B)$ where y is a fresh real variable vector (the same for subsequent calls) and B is the Euclidean norm unit ball.

Returns

A quadruple `(X, aff_y, y, B)` of type (*BaseVariable*, *AffineExpression*, *RealVariable*, *Ball*) such that the two constraints `X.vec == aff_y` and `y << B` combined are equivalent to this one.

Raises

- `NotImplementedError` – If the expression under the norm does not reference exactly one variable or if that variable does not use a trivial vectorization format internally.
- `ValueError` – If the upper bound is not constant, not positive, or if the matrix A is not invertible.

Example

```
>>> import picos
>>> A = picos.Constant("A", [[2, 0],
...                          [0, 1]])
>>> x = picos.RealVariable("x", 2)
>>> P = picos.Problem()
>>> P.set_objective("max", picos.sum(x))
>>> C = P.add_constraint(abs(A*x + 1) <= 10)
>>> _ = P.solve(solver="cvxopt")
>>> print(x)
[ 1.74e+00]
[ 7.94e+00]
>>> Q = picos.Problem()
>>> Q.set_objective("max", picos.sum(x))
>>> x, aff_y, y, B, = C.unit_ball_form
>>> _ = Q.add_constraint(x == aff_y)
>>> _ = Q.add_constraint(y << B)
>>> _ = Q.solve(solver="cvxopt")
>>> print(x)
[ 1.74e+00]
[ 7.94e+00]
>>> round(abs(P.value - Q.value), 4)
0.0
>>> round(y[0]**2 + y[1]**2, 4)
1.0
```

picos.constraints.con_sqnorm

Implementation of *SquaredNormConstraint*.

Classes

class picos.constraints.con_sqnorm.**SquaredNormConstraint**(*squaredNorm*, *upperBound*)

Bases: *Constraint*

Upper bound on a squared Euclidean or Frobenius norm.

class ConicConversion

Bases: *ConstraintConversion*

Upper bound on a squared norm to conic conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*squaredNorm*, *upperBound*)

Construct a *SquaredNormConstraint*.

Parameters

- **squaredNorm** (SquaredNorm) – The squared norm to bound from above.
- **upperBound** (AffineExpression) – Upper bound on the squared norm.

picos.constraints.con_sumexp

Implementation of *SumExponentialsConstraint*.

Classes

class picos.constraints.con_sumexp.**SumExponentialsConstraint**(*theSum*, *upperBound*)

Bases: *Constraint*

Upper bound on a sum of exponentials.

class ConicConversion

Bases: *ConstraintConversion*

Sum of exponentials to exponential cone constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

class LogSumExpConversion

Bases: *ConstraintConversion*

Sum of exponentials to logarithm of the sum constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*theSum*, *upperBound*)

Construct a *SumExponentialsConstraint*.

Parameters

- **theSum** (SumExponentials) – Constrained expression.
- **upperBound** (AffineExpression) – Upper bound on the expression.

property denominator

The y of the sum, or 1.

property lse_representable

Whether this can be converted to a logarithmic constraint.

property numerator

The x of the sum.

picos.constraints.con_sumxtr

Implementation of *SumExtremesConstraint*.

Classes

class picos.constraints.con_sumxtr.**SumExtremesConstraint**(*theSum*, *relation*, *rhs*)

Bases: *Constraint*

Bound on a sum over extreme (eigen)values.

class Conversion

Bases: *ConstraintConversion*

Sum over extremes to LMI/affine constraint conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*theSum*, *relation*, *rhs*)

Construct a *SumExtremesConstraint*.

Parameters

- **theSum** (SumExtremes) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

picos.constraints.con_vecnorm

Implementation of `VectorNormConstraint`.

Classes

class `picos.constraints.con_vecnorm.`**`VectorNormConstraint`**(*norm*, *relation*, *rhs*)

Bases: `Constraint`

Bound on a (generalized) vector p-norm.

class Conversion

Bases: `ConstraintConversion`

Bound on a (generalized) p-norm constraint conversion.

classmethod convert(*con*, *options*)

Implement `convert`.

classmethod predict(*subtype*, *options*)

Implement `predict`.

`__init__`(*norm*, *relation*, *rhs*)

Construct a `VectorNormConstraint`.

Parameters

- **norm** (`Norm`) – Left hand side expression.
- **relation** (`str`) – Constraint relation symbol.
- **rhs** (`AffineExpression`) – Right hand side expression.

picos.constraints.con_wsum

Implementation of `WeightedSumConstraint`.

Classes

class `picos.constraints.con_wsum.`**`WeightedSumConstraint`**(*wsum*, *relation*, *rhs*)

Bases: `Constraint`

Bound on a convex or concave weighted sum of expressions.

class Conversion

Bases: `ConstraintConversion`

Bound on a weighted sum of expressions conversion.

classmethod convert(*con*, *options*)

Implement `convert`.

classmethod predict(*subtype*, *options*)

Implement `predict`.

`__init__`(*wsum*, *relation*, *rhs*)

Construct a `WeightedSumConstraint`.

Parameters

- **wsum** (`WeightedSum`) – Left hand side expression.
- **relation** (`str`) – Constraint relation symbol.

- **rhs** (`AffineExpression`) – Right hand side expression.

picos.constraints.constraint

Backend for constraint type implementations.

Classes

`class picos.constraints.constraint.`**`ConicConstraint`**`(`*typeTerm*, *customString=None*, *printSize=False*`)`

Bases: *`Constraint`*

Base class for constraints with an immediate conic representation.

`abstract property conic_membership_form`

The constraint in conic membership form.

For a conic constraint $Ax \succeq_C b \Leftrightarrow Ax - b \in C$ this is the pair $(Ax - b, C)$ where $Ax - b$ is an affine expression and C a basic cone supported by PICOS.

`property conic_standard_form`

The constraint in conic standard form.

For a conic constraint $Ax \succeq_C b$ this is the triple (Ax, b, C) where Ax is a linear expression, b is a constant, and C a basic cone supported by PICOS.

`property inverse_conic_standard_form`

The constraint in inverse conic standard form.

For a conic constraint $Ax \succeq_C b \Leftrightarrow -Ax \preceq_C -b$ this is the triple $(-Ax, -b, C)$ where Ax is a linear expression, b is a constant, and C a basic cone supported by PICOS.

`class picos.constraints.constraint.`**`Constraint`**`(`*typeTerm*, *customString=None*, *printSize=False*`)`

Bases: `ABC`

Abstract base class for optimization constraints.

Implementations

- need to implement at least the abstract methods `_str`, `_expression_names`, and `_get_slack`,
- need to implement `_get_size`, unless duals are not supported, and
- are supposed to call *`Constraint.__init__`* from within their own implementation of `__init__`.

`__eq__``(`*other*`)`

Whether the unique IDs equal.

`__init__``(`*typeTerm*, *customString=None*, *printSize=False*`)`

Perform basic initialization for *`Constraint`* instances.

Parameters

- **typeTerm** (*str*) – Short string denoting the constraint type.
- **customString** (*str*) – Optional string description.
- **printSize** (*bool*) – Whether to include the constraint's shape in its representation string.

`constring``()`

Return an algebraic string representation of the constraint.

`delete()`

Raise a `NotImplementedError`.

Formerly this would remove the constraint from the single problem it is assigned to, if any.

Deprecated since version 2.0: Both variables and constraints have been decoupled from problems: Both may safely appear in multiple problems but at the same time they do not know which problems they were added to. To remove a constraint from a problem, you have to call its `remove_constraint` method.

`is_decreasing()`

Whether the left side is posed greater or equal than the right.

`is_equality()`

Whether the constraints states an equality.

`is_increasing()`

Whether the left side is posed smaller or equal than the right.

`is_inequality()`

Whether the constraints states an inequality.

`keyconstring()`

Return the regular string representation.

`classmethod make_type(*args, **kwargs)`

Create a detailed constraint type from subtype parameters.

`replace_mutables(new_mutables)`

Make the constraint concern a different set of mutables.

See `replace_mutables` for more.

`EQ = '='`

`GE = '>'`

`LE = '<'`

`property dual`

Value of the constraint's Lagrange dual variable.

`property expressions`

Yield expressions stored with the constraint.

`property id`

The unique ID of the constraint, assigned at creation.

The ID is kept when the constraint is copied via `replace_mutables`, so that the copy can be identified with the original despite pointing to different expressions and mutable objects.

`property mutables`

All mutables referenced by the constraint.

`property parameters`

All parameters referenced by the constraint.

`property size`

Langrange dual variable shape.

The dimensionality of the constraint, more precisely the dimensionality of its Lagrange dual variable, as a pair.

property slack

Value of a slack variable or of the negative constraint violation.

A negative value whose absolute value corresponds to the amount of violation, if the constraint is violated, or a non-negative value that corresponds to the value of a slack variable, otherwise.

property subtype

property type

Detailed type of the constraint.

The detailed type of the constraint, which is suffcient to predict the outcome (detailed types and quantities of auxilary variables and constraints) of any constraint conversion.

property variables

All decision variables referenced by the constraint.

class picos.constraints.constraint.ConstraintConversion

Bases: `ABC`

Recipe for conversion from one constraint to a set of other constraints.

Implementations of this class are defined within the class body of a Constraint implementation to tell PICOS' reformulation framework how that constraint can be reformulated into a number of other constraints and auxiliary variables.

Implementation class names must end in `Conversion`, and in particular may be called just `Conversion`. If for instance *`AbsoluteValueConstraint`* defines *`AffineConversion`*, then the reformulation will be coined `AbsoluteValueToAffineReformulation`. If the conversions was just named `Conversion`, the result would be a class named `AbsoluteValueReformulation`.

`__init__`()

Raise a `TypeError` on instanciation.

abstract classmethod convert(*constraint*, *options*)

Convert a given constraint.

Returns a temporary problem instance that contains auxilary constraints and variables replacing the given constraint.

classmethod dual(*auxVarPrimals*, *auxConDuals*, *options*)

Convert back the dual value of a constraint that was converted.

Given a mapping of auxilary variable names (as named in *`convert`*) to primals and a list of auxilary constraint duals (in the order as the constraints were added in *`convert`*), returns a dual value for the converted constraint.

Raises

`NotImplementedError` – When dual format not decided upon or not known. This will be caught by the reformulation's backward method.

abstract classmethod predict(*subtype*, *options*)

Predict the outcome of a constraint conversion.

Parameters

- **subtype** (*`object`*) – A hashable object as could be returned by the `_subtype` method of the parent constraint implementation.
- **options** (`Options`) – Solution options to assume used.

Yields

Records to be added to a problem footprint when an instance of the parent constraint with the given subtype is converted according to this conversion.

class picos.constraints.constraint.**ConstraintType**(*theClass*, *subtype*)

Bases: `DetailedType`

Container for a pair of constraint class type and constraint subtype.

picos.constraints.uncertain

Constraint types with an explicit representation of data uncertainty.

Classes

class picos.constraints.uncertain.**BallUncertainNormConstraint**

See *picos.constraints.uncertain.ucon_ball_norm.BallUncertainNormConstraint*.

class picos.constraints.uncertain.**ConicallyUncertainAffineConstraint**

See *picos.constraints.uncertain.ucon_conic_aff.ConicallyUncertainAffineConstraint*.

class picos.constraints.uncertain.**MomentAmbiguousExtremumAffineConstraint**

See *picos.constraints.uncertain.ucon_mom_pwl.MomentAmbiguousExtremumAffineConstraint*.

class picos.constraints.uncertain.**MomentAmbiguousSquaredNormConstraint**

See *picos.constraints.uncertain.ucon_mom_sqnorm.MomentAmbiguousSquaredNormConstraint*.

class picos.constraints.uncertain.**ScenarioUncertainConicConstraint**

See *picos.constraints.uncertain.ucon_scen_conic.ScenarioUncertainConicConstraint*.

class picos.constraints.uncertain.**WassersteinAmbiguousExtremumAffineConstraint**

See *picos.constraints.uncertain.ucon_ws_pwl.WassersteinAmbiguousExtremumAffineConstraint*.

class picos.constraints.uncertain.**WassersteinAmbiguousSquaredNormConstraint**

See *picos.constraints.uncertain.ucon_ws_sqnorm.WassersteinAmbiguousSquaredNormConstraint*.

picos.constraints.uncertain.ucon_ball_norm

Implements *BallUncertainNormConstraint*.

Classes

class picos.constraints.uncertain.ucon_ball_norm.**BallUncertainNormConstraint**(*norm*, *upper_bound*)

Bases: `Constraint`

An (uncertain) upper bound on a norm with unit ball uncertainty.

class RobustConversion

Bases: `ConstraintConversion`

Robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement `convert`.

Conversion recipe and variable names based on the book *Robust Optimization* (Ben-Tal, El Ghaoui, Nemirovski, 2009).

classmethod predict(*subtype*, *options*)

Implement `predict`.

`__init__`(*norm*, *upper_bound*)

Construct a *`BallUncertainNormConstraint`*.

Parameters

- **`norm`** (`UncertainNorm`) – Uncertain norm that is bounded from above.
- **`upper_bound`** (`AffineExpression` *or* `UncertainAffineExpression`) – (Uncertain) upper bound on the norm.

`property ne`

The uncertain affine expression under the norm.

picos.constraints.uncertain.ucon_conic_aff

Implements *`ConicallyUncertainAffineConstraint`*.

Classes

`class picos.constraints.uncertain.ucon_conic_aff.`**`ConicallyUncertainAffineConstraint`**(*le0*)

Bases: *`Constraint`*

A bound on an affine expression with conic uncertainty.

`class RobustConversion`

Bases: *`ConstraintConversion`*

Robust counterpart conversion.

`classmethod convert`(*con*, *options*)

Implement *`convert`*.

Conversion recipe and variable names based on the book *Robust Optimization* (Ben-Tal, El Ghaoui, Nemirovski, 2009).

`classmethod predict`(*subtype*, *options*)

Implement *`predict`*.

`__init__`(*le0*)

Construct an *`ConicallyUncertainAffineConstraint`*.

Parameters

`le0` (`UncertainAffineExpression`) – Uncertain expression constrained to be at most zero.

picos.constraints.uncertain.ucon_mom_pwl

Implements *`MomentAmbiguousExtremumAffineConstraint`*.

Classes

class picos.constraints.uncertain.ucon_mom_pwl.**MomentAmbiguousExtremumAffineConstraint**(*extremum*, *relation*, *rhs*)

Bases: *Constraint*

A bound on a moment-ambiguous expected value of a piecewise function.

class DistributionallyRobustConversion

Bases: *ConstraintConversion*

Distributionally robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*extremum*, *relation*, *rhs*)

Construct a *MomentAmbiguousExtremumAffineConstraint*.

Parameters

- **extremum** (RandomExtremumAffine) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

property maximum_form

The constraint posed as an upper bound on an expected maximum.

picos.constraints.uncertain.ucon_mom_sqnorm

Implements *MomentAmbiguousSquaredNormConstraint*.

Classes

class picos.constraints.uncertain.ucon_mom_sqnorm.**MomentAmbiguousSquaredNormConstraint**(*sqnorm*, *upper_bound*)

Bases: *Constraint*

A bound on a moment-ambiguous expected value of a squared norm.

class DistributionallyRobustConversion

Bases: *ConstraintConversion*

Distributionally robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*sqnorm*, *upper_bound*)

Construct a *MomentAmbiguousSquaredNormConstraint*.

Parameters

- **sqnorm** (UncertainSquaredNorm) – Uncertain squared norm to upper bound the expectation of.
- **upper_bound** (AffineExpression) – Upper bound on the expected value.

picos.constraints.uncertain.ucon_scen_conic

Implements *ScenarioUncertainConicConstraint*.

Classes

class picos.constraints.uncertain.ucon_scen_conic.**ScenarioUncertainConicConstraint**(*element*, *cone*)

Bases: *Constraint*

Conic constraint with scenario uncertainty.

class RobustConversion

Bases: *ConstraintConversion*

Robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*element*, *cone*)

Construct a *ScenarioUncertainConicConstraint*.

Parameters

- **element** (UncertainAffineExpression) – Uncertain expression constrained to be in the cone.
- **cone** (Cone) – The cone that the uncertain expression is constrained to.

picos.constraints.uncertain.ucon_ws_pwl

Implements *WassersteinAmbiguousExtremumAffineConstraint*.

Classes

class picos.constraints.uncertain.ucon_ws_pwl.**WassersteinAmbiguousExtremumAffineConstraint**(*extremum*, *relation*, *rhs*)

Bases: *Constraint*

A bound on a W_1-ambiguous expected value of a piecewise function.

class DistributionallyRobustConversion

Bases: *ConstraintConversion*

Distributionally robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*extremum*, *relation*, *rhs*)

Construct a *WassersteinAmbiguousExtremumAffineConstraint*.

Parameters

- **extremum** (RandomExtremumAffine) – Left hand side expression.
- **relation** (*str*) – Constraint relation symbol.
- **rhs** (AffineExpression) – Right hand side expression.

property maximum_form

The constraint posed as an upper bound on an expected maximum.

picos.constraints.uncertain.ucon_ws_sqnorm

Implements *WassersteinAmbiguousSquaredNormConstraint*.

Classes

class picos.constraints.uncertain.ucon_ws_sqnorm.**WassersteinAmbiguousSquaredNormConstraint**(*sqnorm*, *upper_bound*)

Bases: *Constraint*

A bound on a Wasserstein-ambiguous expected value of a squared norm.

class DistributionallyRobustConversion

Bases: *ConstraintConversion*

Distributionally robust counterpart conversion.

classmethod convert(*con*, *options*)

Implement *convert*.

classmethod predict(*subtype*, *options*)

Implement *predict*.

__init__(*sqnorm*, *upper_bound*)

Construct a *WassersteinAmbiguousSquaredNormConstraint*.

Parameters

- **sqnorm** (UncertainSquaredNorm) – Uncertain squared norm to upper bound the expectation of.
- **upper_bound** (AffineExpression) – Upper bound on the expected value.

6.1.4 picos.containers

Domain-specific container types.

Classes

class picos.containers.**DetailedType**(*theClass*, *subtype*)

Bases: `object`

Container for a pair of Python class and subtype.

A container for a pair of a python class type and some logical subtype data structure, together called a detailed type.

A detailed type is used when the mathematical type of an object must be distinguished more precisely than at the level of the python classes used to represent such mathematical objects. For instance, a single python class would be used for a type of expressions of varying dimensionality and subtypes would be used to distinguish further based on dimension.

Instances of this class are treated exceptionally when used as a label of a `RecordTree`: They are expanded into the class and the subtype as two seperate labels, making it convenient to store detailed types in trees.

__add__(*other*)

__eq__(*other*)

Return self==value.

__init__(*theClass*, *subtype*)

Construct a `DetailedType`.

Parameters

- **theClass** (`type`) – The Python class part of the detailed type.
- **subtype** (`object`) – Additional type information.

__radd__(*other*)

equals(*other*)

Whether two detailed types are the same.

class picos.containers.**OrderedSet**(*iterable=()*)

Bases: `MutableSet`

A set that remembers its insertion order.

```
>>> from picos.containers import OrderedSet as oset
>>> o = oset([4, 3, 2, 1]); o
OrderedSet([4, 3, 2, 1])
>>> 3 in o
True
>>> o.update([5, 4, 3]); o
OrderedSet([4, 3, 2, 1, 5])
>>> list(o)
[4, 3, 2, 1, 5]
```

__init__(*iterable=()*)

Intialize the ordered set.

Parameters

iterable – Iterable to take initial elements from.

add(*element*)

Add an element to the set.

clear()

Clear the set.

discard(*element*)

Discard an element from the set.

If the element is not contained, do nothing.

update(**iterables*)

Update the set with elements from a number of iterables.

property difference

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

property difference_update

Remove all elements of another set from this set.

property intersection

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

property intersection_update

Update a set with the intersection of itself and another.

property issubset

Test whether every element in the set is in other.

property issuperset

Test whether every element in other is in the set.

property symmetric_difference

Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

property symmetric_difference_update

Update a set with the symmetric difference of itself and another.

property union

Return the union of sets as a new set.

(i.e. all elements that are in either set.)

class picos.containers.**RecordTree**(*recordsOrDict=()*, *addValues=False*, *freeze=True*)

Bases: `object`

Labeled tree for storing records.

An immutable labeled tree with values at the leaf nodes, where labels and values are arbitrary hashable python objects.

An n-tuple whose first (n-1) elements are labels and whose last element is a value is called a record. Thus, every path from the root node to a leaf node represents one record.

`DetailedType` labels are treated exceptionally: They are expanded into the class and the subtype as two seperate labels.

class ALL

Bases: *RecordTreeToken*

Special *RecordTree* value: Any subtrees.

A special value that behaves as an arbitrary subtree during subtree checks.

class NONE

Bases: *RecordTreeToken*

Special *RecordTree* value: No subtrees.

If inserted at some (inner) node of the tree, the whole subtree starting at that node is deleted. If that node's parent node has no other children, then the parent node is deleted as well. This process is repeated iteratively up to the root node, which is never deleted.

This is the only value that may be inserted at an inner node.

This value cannot itself be stored in the tree as its insertion is always read as a deletion.

__eq__(*other*)

Return self==value.

__ge__(*other*)

Perform entrywise greater-or-equal-than comparison.

Each left hand side path must be present on the right hand side, and the associated left hand side value must compare greater-or-equal-than the right hand side value.

__init__(*recordsOrDict=()*, *addValues=False*, *freeze=True*)

Construct a *RecordTree*.

Parameters

- **recordsOrDict** (*dict or list(tuple)*) – Data stored in the tree.
- **addValues** (*bool or list(tuple)*) – Add the (numeric) values of records stored in the same place in the tree, instead of replacing the value. If this is exactly a list of path tuples (precise types are required), then add values only for records below any of these paths instead. In either case, resulting values of zero are not stored in the tree.
- **freeze** (*bool*) – Whether to transform mutable labels and values into hashable ones.

__le__(*other*)

Perform entrywise lower-or-equal-than comparison.

Each left hand side path must be present on the right hand side, and the associated left hand side value must compare lower-or-equal-than the right hand side value.

__lshift__(*other*)

Perform subtree comparison.

Each left hand side path must be present on the right hand side. If the special *ALL* type is found as a value in the right hand side tree, it is treated as “all possible subtrees”. All other values are not considered.

copy()

Create a shallow copy; the tree is copied, the values are not.

get(*path*)

Return an empty *RecordTree* if the path does not exist.

mismatch(*other*)

A subtree of `self` that renders `self << other` `False`.

Returns RecordTree

The smallest subtree `T` of `self` such that `self` without the records in `T` is a subtree of `other`. The returned tree is a direct instance of the *RecordTree* base class.

updated(*recordsOrDict*, *addValues=False*)

Create a shallow copy with modified records.

Example

```
>>> from picos.modeling.footprint import RecordTree as T
>>> t = T({(1, 1, 1): 3, (1, 1, 2): 4, (1, 2, 1): 5}); t
RecordTree({(1, 1, 1): 3, (1, 1, 2): 4, (1, 2, 1): 5})
>>> t.updated({(1, 1, 1): "a", (2, 2): "b"}) # Change or add a record.
RecordTree({(1, 1, 1): 'a', (1, 1, 2): 4, (1, 2, 1): 5, (2, 2): 'b'})
>>> t.updated({(1,1,1): T.NONE}) # Delete a single record.
RecordTree({(1, 1, 2): 4, (1, 2, 1): 5})
>>> t.updated({(1,1): T.NONE}) # Delete multiple records.
RecordTree({(1, 2, 1): 5})
>>> t.updated([(1, 1, 1, T.NONE), (1, 1, 1, 1, 6)]) # Delete, then add.
RecordTree({(1, 1, 2): 4, (1, 1, 1, 1): 6, (1, 2, 1): 5})
>>> try: # Not possible to implicitly turn a leaf into an inner node.
...     t.updated([(1, 1, 1, 1, 6)])
... except LookupError as error:
...     print(error)
Can't set value '6' at '(1, 1, 1, 1)': Path already contains a leaf.
```

property dict

Return the tree as a read-only, tuple-indexed dictionary view.

Every key/value pair of the returned dictionary is a record.

property items

Return an iterator over path/value pairs representing records.

property paths

Return an iterator over paths, each representing one record.

property records

Return an iterator over tuples, each representing one record.

property set

Return a frozen set of tuples, each representing one record.

property text

Return the full tree as a multiline string.

class picos.containers.**RecordTreeToken**

Bases: `object`

Base class for special *RecordTree* value tokens.

__init__()

Raise a `TypeError` on instanciation.

class picos.containers.**frozendict**

Bases: `dict`

An immutable, hashable dictionary.

copy()

Since *frozendict* are immutable, returns self.

classmethod fromkeys(*iterable*, *value=None*)

Overwrite `dict.fromkeys`.

property clear

property pop

If the key is not found, return the default if given; otherwise, raise a KeyError.

property popitem

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

property setdefault

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

property update

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

6.1.5 picos.expressions

Mathematical expression types.

Exceptions

exception picos.expressions.**IntractableWorstCase**

See *picos.expressions.uncertain.uexpression.IntractableWorstCase*.

exception picos.expressions.**NotValued**

See *picos.valuable.NotValued*.

exception picos.expressions.**PredictedFailure**

See *picos.expressions.expression.PredictedFailure*.

Classes

class picos.expressions.**AffineExpression**

See *picos.expressions.exp_affine.AffineExpression*.

class picos.expressions.**Ball**

See *picos.expressions.set_ball.Ball*.

class picos.expressions.**BaseVariable**

See *picos.expressions.variables.BaseVariable*.

class picos.expressions.**BiaffineExpression**

See *picos.expressions.exp_biaffine.BiaffineExpression*.

class picos.expressions.**BinaryVariable**

See *picos.expressions.variables.BinaryVariable*.

class picos.expressions.**ComplexAffineExpression**

See *picos.expressions.exp_affine.ComplexAffineExpression*.

class picos.expressions.**ComplexVariable**

See *picos.expressions.variables.ComplexVariable*.

class picos.expressions.**Cone**

See *picos.expressions.cone.Cone*.

class picos.expressions.**ConicPerturbationSet**

See *picos.expressions.uncertain.pert_conic.ConicPerturbationSet*.

class picos.expressions.**DetRootN**

See *picos.expressions.exp_detrootn.DetRootN*.

class picos.expressions.**Ellipsoid**

See *picos.expressions.set_ellipsoid.Ellipsoid*.

class picos.expressions.**Entropy**

See *picos.expressions.exp_entropy.Entropy*.

class picos.expressions.**ExponentialCone**

See *picos.expressions.cone_expcone.ExponentialCone*.

class picos.expressions.**Expression**

See *picos.expressions.expression.Expression*.

class picos.expressions.**ExpressionType**

See *picos.expressions.expression.ExpressionType*.

class picos.expressions.**Extremum**

See *picos.expressions.exp_extremum.Extremum*.

class picos.expressions.**GeometricMean**

See *picos.expressions.exp_geomean.GeometricMean*.

class picos.expressions.**HermitianVariable**

See *picos.expressions.variables.HermitianVariable*.

class picos.expressions.**IntegerVariable**

See *picos.expressions.variables.IntegerVariable*.

class picos.expressions.**LogSumExp**

See *picos.expressions.exp_logsumexp.LogSumExp*.

class picos.expressions.**Logarithm**

See *picos.expressions.exp_logarithm.Logarithm*.

class picos.expressions.**LowerTriangularVariable**

See *picos.expressions.variables.LowerTriangularVariable*.

class picos.expressions.**MatrixGeometricMean**

See *picos.expressions.exp_mtxgeomean.MatrixGeometricMean*.

class picos.expressions.**MaximumConvex**

See *picos.expressions.exp_extremum.MaximumConvex*.

class picos.expressions.**MinimumConcave**

See *picos.expressions.exp_extremum.MinimumConcave*.

class picos.expressions.**MomentAmbiguitySet**

See *picos.expressions.uncertain.pert_moment.MomentAmbiguitySet*.

class picos.expressions.**Mutable**

See *picos.expressions.mutable.Mutable*.

class picos.expressions.**NegativeEntropy**

See *picos.expressions.exp_entropy.NegativeEntropy*.

class picos.expressions.**NegativeQuantumEntropy**

See *picos.expressions.exp_quantentr.NegativeQuantumEntropy*.

class picos.expressions.**NonnegativeOrthant**

See *picos.expressions.cone_nno.NonnegativeOrthant*.

class picos.expressions.**Norm**

See *picos.expressions.exp_norm.Norm*.

class picos.expressions.**NuclearNorm**

See *picos.expressions.exp_nucnorm.NuclearNorm*.

class picos.expressions.**OperatorRelativeEntropy**

See *picos.expressions.exp_oprelentr.OperatorRelativeEntropy*.

class picos.expressions.**Perturbation**

See *picos.expressions.uncertain.perturbation.Perturbation*.

class picos.expressions.**PerturbationUniverse**

See *picos.expressions.uncertain.perturbation.PerturbationUniverse*.

class picos.expressions.**PositiveSemidefiniteCone**

See *picos.expressions.cone_psd.PositiveSemidefiniteCone*.

class picos.expressions.**PowerTrace**

See *picos.expressions.exp_powtrace.PowerTrace*.

class picos.expressions.**ProductCone**

See *picos.expressions.cone_product.ProductCone*.

class picos.expressions.**QuadraticExpression**

See *picos.expressions.exp_quadratic.QuadraticExpression*.

class picos.expressions.**QuantumConditionalEntropy**

See *picos.expressions.exp_quantcondentr.QuantumConditionalEntropy*.

class picos.expressions.**QuantumEntropy**

See *picos.expressions.exp_quantentr.QuantumEntropy*.

class picos.expressions.**QuantumKeyDistribution**

See *picos.expressions.exp_quantkeydist.QuantumKeyDistribution*.

class picos.expressions.**RandomExtremumAffine**

See *picos.expressions.uncertain.uexp_rand_pwl.RandomExtremumAffine*.

class picos.expressions.**RandomMaximumAffine**

See *picos.expressions.uncertain.uexp_rand_pwl.RandomMaximumAffine*.

class picos.expressions.**RandomMinimumAffine**

See *picos.expressions.uncertain.uexp_rand_pwl.RandomMinimumAffine*.

class picos.expressions.**RealVariable**

See *picos.expressions.variables.RealVariable*.

class picos.expressions.**RotatedSecondOrderCone**

See *picos.expressions.cone_rsoc.RotatedSecondOrderCone*.

class picos.expressions.**Samples**

See *picos.expressions.samples.Samples*.

class picos.expressions.**ScenarioPerturbationSet**

See *picos.expressions.uncertain.pert_scenario.ScenarioPerturbationSet*.

class picos.expressions.**SecondOrderCone**

See *picos.expressions.cone_soc.SecondOrderCone*.

class picos.expressions.**Set**
See *picos.expressions.set.Set*.

class picos.expressions.**SetType**
See *picos.expressions.set.SetType*.

class picos.expressions.**Simplex**
See *picos.expressions.set_simplex.Simplex*.

class picos.expressions.**SkewSymmetricVariable**
See *picos.expressions.variables.SkewSymmetricVariable*.

class picos.expressions.**SpectralNorm**
See *picos.expressions.exp_specnorm.SpectralNorm*.

class picos.expressions.**SquaredNorm**
See *picos.expressions.exp_sqnorm.SquaredNorm*.

class picos.expressions.**SumExponentials**
See *picos.expressions.exp_sumexp.SumExponentials*.

class picos.expressions.**SumExtremes**
See *picos.expressions.exp_sumxtr.SumExtremes*.

class picos.expressions.**SymmetricVariable**
See *picos.expressions.variables.SymmetricVariable*.

class picos.expressions.**TheField**
See *picos.expressions.cone_trivial.TheField*.

class picos.expressions.**TrMatrixGeometricMean**
See *picos.expressions.exp_mtxgeomean.TrMatrixGeometricMean*.

class picos.expressions.**TrOperatorRelativeEntropy**
See *picos.expressions.exp_oprelentr.TrOperatorRelativeEntropy*.

class picos.expressions.**UncertainAffineExpression**
See *picos.expressions.uncertain.uexp_affine.UncertainAffineExpression*.

class picos.expressions.**UncertainExpression**
See *picos.expressions.uncertain.uexpression.UncertainExpression*.

class picos.expressions.**UncertainNorm**
See *picos.expressions.uncertain.uexp_norm.UncertainNorm*.

class picos.expressions.**UncertainSquaredNorm**
See *picos.expressions.uncertain.uexp_sqnorm.UncertainSquaredNorm*.

class picos.expressions.**UnitBallPerturbationSet**
See *picos.expressions.uncertain.pert_conic.UnitBallPerturbationSet*.

class picos.expressions.**UpperTriangularVariable**
See *picos.expressions.variables.UpperTriangularVariable*.

class picos.expressions.**WassersteinAmbiguitySet**
See *picos.expressions.uncertain.pert_wasserstein.WassersteinAmbiguitySet*.

class picos.expressions.**WeightedSum**
See *picos.expressions.exp_wsum.WeightedSum*.

class picos.expressions.**ZeroSpace**
See *picos.expressions.cone_trivial.ZeroSpace*.

Functions

picos.expressions.**Constant**()

See *picos.expressions.exp_affine.Constant*.

picos.expressions.**ball**()

See *picos.expressions.algebra.ball*.

picos.expressions.**block**()

See *picos.expressions.algebra.block*.

picos.expressions.**detrootn**()

See *picos.expressions.algebra.detrootn*.

picos.expressions.**diag**()

See *picos.expressions.algebra.diag*.

picos.expressions.**diag_vect**()

See *picos.expressions.algebra.diag_vect*.

picos.expressions.**exp**()

See *picos.expressions.algebra.exp*.

picos.expressions.**expcone**()

See *picos.expressions.algebra.expcone*.

picos.expressions.**flow_Constraint**()

See *picos.expressions.algebra.flow_Constraint*.

picos.expressions.**geomean**()

See *picos.expressions.algebra.geomean*.

picos.expressions.**kldiv**()

See *picos.expressions.algebra.kldiv*.

picos.expressions.**kron**()

See *picos.expressions.algebra.kron*.

picos.expressions.**kullback_leibler**()

See *picos.expressions.algebra.kullback_leibler*.

picos.expressions.**lambda_max**()

See *picos.expressions.algebra.lambda_max*.

picos.expressions.**lambda_min**()

See *picos.expressions.algebra.lambda_min*.

picos.expressions.**log**()

See *picos.expressions.algebra.log*.

picos.expressions.**logsumexp**()

See *picos.expressions.algebra.logsumexp*.

picos.expressions.**lse**()

See *picos.expressions.algebra.lse*.

picos.expressions.**maindiag**()

See *picos.expressions.algebra.maindiag*.

picos.expressions.**max**()

See *picos.expressions.algebra.max*.

picos.expressions.**min**()

See *picos.expressions.algebra.min*.

picos.expressions.**mtxgeomean**()

See *picos.expressions.algebra.oprelentr*.

picos.expressions.**new_param**()

See *picos.expressions.algebra.new_param*.

picos.expressions.**no_refinement**()

See *picos.expressions.expression.no_refinement*.

picos.expressions.**norm**()

See *picos.expressions.algebra.norm*.

picos.expressions.**oprelentr**()

See *picos.expressions.algebra.oprelentr*.

picos.expressions.**partial_trace**()

See *picos.expressions.algebra.partial_trace*.

picos.expressions.**partial_transpose**()

See *picos.expressions.algebra.partial_transpose*.

picos.expressions.**quantcondentr**()

See *picos.expressions.algebra.quantcondentr*.

picos.expressions.**quantentr**()

See *picos.expressions.algebra.quantentr*.

picos.expressions.**quantkeydist**()

See *picos.expressions.algebra.quantkeydist*.

picos.expressions.**quantrelentr**()

See *picos.expressions.algebra.quantrelentr*.

picos.expressions.**rsoc**()

See *picos.expressions.algebra.rsoc*.

picos.expressions.**simplex**()

See *picos.expressions.algebra.simplex*.

picos.expressions.**soc**()

See *picos.expressions.algebra.soc*.

picos.expressions.**sum**()

See *picos.expressions.algebra.sum*.

picos.expressions.**sum_k_largest**()

See *picos.expressions.algebra.sum_k_largest*.

picos.expressions.**sum_k_largest_lambda**()

See *picos.expressions.algebra.sum_k_largest_lambda*.

picos.expressions.**sum_k_smallest**()

See *picos.expressions.algebra.sum_k_smallest*.

picos.expressions.**sum_k_smallest_lambda**()

See *picos.expressions.algebra.sum_k_smallest_lambda*.

picos.expressions.**sumexp**()

See *picos.expressions.algebra.sumexp*.

picos.expressions.**trace**()

See *picos.expressions.algebra.trace*.

picos.expressions.**tracepow**()

See *picos.expressions.algebra.tracepow*.

picos.expressions.**truncated_simplex**()

See *picos.expressions.algebra.truncated_simplex*.

Objects

picos.expressions.**CONTINUOUS_VARTYPES**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

Default value

```
(<class 'picos.expressions.variables.RealVariable'>,
 <class 'picos.expressions.variables.ComplexVariable'>,
 <class 'picos.expressions.variables.SymmetricVariable'>,
 <class 'picos.expressions.variables.SkewSymmetricVariable'>,
 <class...
```

picos.expressions.**I**

Create an identity matrix.

Example

```
>>> from picos import I
>>> print(I(3))
[ 1.00e+00     0         0    ]
[    0      1.00e+00     0    ]
[    0         0      1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c9e0>
```

picos.expressions.**J**

Create a matrix of all ones.

Example

```
>>> from picos import J
>>> print(J(2, 3))
[ 1.00e+00  1.00e+00  1.00e+00]
[ 1.00e+00  1.00e+00  1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655ca90>
```

`picos.expressions.O`

Create a zero matrix.

Example

```
>>> from picos import O
>>> print(O(2, 3))
[0 0 0]
[0 0 0]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c930>
```

picos.expressions.algebra

Implements functions that create or modify algebraic expressions.

Functions

`picos.expressions.algebra.`**`ball`**`(`*`*args`*`, `*`**kwargs`*`)`

Legacy shorthand for `Ball`.

Deprecated since version 2.0: Use `picos.Ball` instead.

`picos.expressions.algebra.`**`block`**`(`*nested*, *shapes=None*, *name=None*`)`

Create an affine block matrix expression.

Given a two-level nested iterable container (e.g. a list of lists) of PICOS affine expressions or constant data values or a mix thereof, this creates an affine block matrix where each inner container represents one block row and each expression or constant represents one block.

Blocks that are given as PICOS expressions are never reshaped or broadcasted. Their shapes must already be consistent. Blocks that are given as constant data values are reshaped or broadcasted as necessary **to match existing PICOS expressions**. This means you can specify blocks as e.g. `"I"` or `0` and PICOS will load them as matrices with the smallest shape that is consistent with other blocks given as PICOS expressions.

Since constant data values are not reshaped or broadcasted with respect to each other, the `shapes` parameter allows a manual clarification of block shapes. It must be consistent with the shapes of blocks given as PICOS expressions (they are still not reshaped or broadcasted).

Parameters

- **`nested`** (*`tuple(tuple)` or `list(list)`*) – The blocks.
- **`shapes`** (*`tuple(tuple)` or `list(list)`*) – A pair `(rows, columns)` where `rows` defines the number of rows for each block row and `columns` defines the number of columns for each block column. You can put a `0` or `None` as a wildcard.
- **`name`** (*`str`*) – Name or string description of the resulting block matrix. If `None`, a descriptive string will be generated.

Example

```
>>> from picos import block, Constant, RealVariable
>>> C = Constant("C", range(6), (3, 2))
>>> d = Constant("d", 0.5, 2)
>>> x = RealVariable("x", 3)
>>> A = block([[ C,   x ],
...            ["I", d ]]); A
```

(continues on next page)

(continued from previous page)

```
<5×3 Real Affine Expression: [C, x; I, d]>
>>> x.value = [60, 70, 80]
>>> print(A)
[ 0.00e+00  3.00e+00  6.00e+01]
[ 1.00e+00  4.00e+00  7.00e+01]
[ 2.00e+00  5.00e+00  8.00e+01]
[ 1.00e+00  0.00e+00  5.00e-01]
[ 0.00e+00  1.00e+00  5.00e-01]
>>> B = block([[ C,   x ],  # With a shape hint.
...            ["I", 0 ]], shapes=((3, 2), (2, 1))); B
<5×3 Real Affine Expression: [C, x; I, 0]>
>>> print(B)
[ 0.00e+00  3.00e+00  6.00e+01]
[ 1.00e+00  4.00e+00  7.00e+01]
[ 2.00e+00  5.00e+00  8.00e+01]
[ 1.00e+00  0.00e+00  0.00e+00]
[ 0.00e+00  1.00e+00  0.00e+00]
```

`picos.expressions.algebra.`**`detrootn`**`(`**args*, ***kwargs*`)`

Legacy shorthand for *`DetRootN`*.

Deprecated since version 2.0: Use *`picos.DetRootN`* instead.

`picos.expressions.algebra.`**`diag`**`(`*x*, *n=1*`)`

Form a diagonal matrix from the column-major vectorization of x.

If $n \neq 1$, then the vectorization is repeated n times.

`picos.expressions.algebra.`**`diag_vect`**`(`*x*`)`

Extract the diagonal of x as a column vector.

Deprecated since version 2.0: Use *`picos.maindiag`* instead.

`picos.expressions.algebra.`**`exp`**`(`*x*`)`

Denote the exponential.

`picos.expressions.algebra.`**`expcone`**`(`**args*, ***kwargs*`)`

Shorthand for *`ExponentialCone`*.

`picos.expressions.algebra.`**`flow_Constraint`**`(`**args*, ***kwargs*`)`

Legacy shorthand for *`FlowConstraint`*.

Deprecated since version 2.0: Use *`picos.FlowConstraint`* instead.

`picos.expressions.algebra.`**`geomean`**`(`**args*, ***kwargs*`)`

Shorthand for *`GeometricMean`*.

`picos.expressions.algebra.`**`kldiv`**`(`**args*, ***kwargs*`)`

Shorthand for *`NegativeEntropy`*.

`picos.expressions.algebra.`**`kron`**`(`*x*, *y*`)`

Denote the kronecker product.

Deprecated since version 2.2: Ensure that one operand is a PICOS expression and use infix @ instead.

`picos.expressions.algebra.`**`kullback_leibler`**`(`**args*, ***kwargs*`)`

Legacy shorthand for *`NegativeEntropy`*.

Deprecated since version 2.0: Use *`picos.kldiv`* instead.

picos.expressions.algebra.**lambda_max**(*x*)

Wrapper for *SumExtremes*.

Sets k = 1, largest = True and eigenvalues = True.

Example

```
>>> from picos import SymmetricVariable, lambda_max
>>> X = SymmetricVariable("X", 5)
>>> lambda_max(X)
<Largest Eigenvalue: λ_max(X)>
>>> lambda_max(X) <= 2
<Largest Eigenvalue Constraint: λ_max(X) ≤  2>
```

picos.expressions.algebra.**lambda_min**(*x*)

Wrapper for *SumExtremes*.

Sets k = 1, largest = False and eigenvalues = True.

Example

```
>>> from picos import SymmetricVariable, lambda_min
>>> X = SymmetricVariable("X", 5)
>>> lambda_min(X)
<Smallest Eigenvalue: λ_min(X)>
>>> lambda_min(X) >= 2
<Smallest Eigenvalue Constraint: λ_min(X) ≥  2>
```

picos.expressions.algebra.**log**(*x*)

Denote the natural logarithm.

picos.expressions.algebra.**logsumexp**(**args*, ***kwargs*)

Legacy shorthand for *LogSumExp*.

Deprecated since version 2.0: Use *picos.lse* instead.

picos.expressions.algebra.**lse**(**args*, ***kwargs*)

Shorthand for *LogSumExp*.

picos.expressions.algebra.**maindiag**(*x*)

Extract the diagonal of x as a column vector.

picos.expressions.algebra.**max**(*lst*)

Denote the maximum over a collection of convex scalar expressions.

If instead of a collection of expressions only a single multidimensional affine expression is given, this denotes its largest element instead.

If some individual expressions are uncertain and their uncertainty is not of stochastic but of worst-case nature (robust optimization), then the maximum implicitly goes over their perturbation parameters as well.

Parameters

lst (*list or tuple or* AffineExpression) – A list of convex expressions or a single affine expression.

Example

```
>>> from picos import RealVariable, max, sum
>>> x = RealVariable("x", 5)
>>> max(x)
<Largest Element: max(x)>
>>> max(x) <= 2  # The same as x <= 2.
<Largest Element Constraint: max(x) ≤  2>
```

(continues on next page)

(continued from previous page)

```
>>> max([sum(x), abs(x)])
<Maximum of Convex Functions: max(∑(x), ‖x‖)>
>>> max([sum(x), abs(x)]) <= 2  # Both must be <= 2.
<Maximum of Convex Functions Constraint: max(∑(x), ‖x‖) ≤ 2>
>>> from picos.uncertain import UnitBallPerturbationSet
>>> z = UnitBallPerturbationSet("z", 5).parameter
>>> max([sum(x), x.T*z])  # Also maximize over z.
<Maximum of Convex Functions: max(∑(x), max_z xᵀ·z)>
```

`picos.expressions.algebra.`**`min`**(*lst*)

Denote the minimum over a collection of concave scalar expressions.

If instead of a collection of expressions only a single multidimensional affine expression is given, this denotes its smallest element instead.

If some individual expressions are uncertain and their uncertainty is not of stochastic but of worst-case nature (robust optimization), then the minimum implicitly goes over their perturbation parameters as well.

Parameters

lst (*list or tuple or AffineExpression*) – A list of concave expressions or a single affine expression.

Example

```
>>> from picos import RealVariable, min, sum
>>> x = RealVariable("x", 5)
>>> min(x)
<Smallest Element: min(x)>
>>> min(x) >= 2  # The same as x >= 2.
<Smallest Element Constraint: min(x) ≥ 2>
>>> min([sum(x), -x[0]**2])
<Minimum of Concave Functions: min(∑(x), -x[0]²)>
>>> min([sum(x), -x[0]**2]) >= 2  # Both must be >= 2.
<Minimum of Concave Functions Constraint: min(∑(x), -x[0]²) ≥ 2>
>>> from picos.uncertain import UnitBallPerturbationSet
>>> z = UnitBallPerturbationSet("z", 5).parameter
>>> min([sum(x), x.T*z])  # Also minimize over z.
<Minimum of Concave Functions: min(∑(x), min_z xᵀ·z)>
```

`picos.expressions.algebra.`**`mtxgeomean`**(**args*, ***kwargs*)

Shorthand for *MatrixGeometricMean*.

`picos.expressions.algebra.`**`new_param`**(*name*, *value*)

Create a constant or a list or dict or tuple thereof.

Deprecated since version 2.0: Use *picos.Constant* instead.

`picos.expressions.algebra.`**`norm`**(**args*, ***kwargs*)

Legacy shorthand for *Norm*.

Deprecated since version 2.0: Use *picos.Norm* instead.

`picos.expressions.algebra.`**`oprelentr`**(**args*, ***kwargs*)

Shorthand for *OperatorRelativeEntropy*.

`picos.expressions.algebra.`**`partial_trace`**(*x*, *subsystems=0*, *dimensions=2*, *k=None*, *dim=None*)

See *exp_biaffine.BiaffineExpression.partial_trace*.

The parameters *k* and *dim* are for backwards compatibility.

`picos.expressions.algebra.`**`partial_transpose`**`(`*x*, *subsystems=0*, *dimensions=2*, *k=None*, *dim=None*`)`

See *`exp_biaffine.BiaffineExpression.partial_transpose`*.

The parameters *k* and *dim* are for backwards compatibility.

`picos.expressions.algebra.`**`quantcondentr`**`(`**args*, ***kwargs*`)`

Shorthand for *`QuantumConditionalEntropy`*.

`picos.expressions.algebra.`**`quantentr`**`(`**args*, ***kwargs*`)`

Shorthand for *`QuantumEntropy`*.

`picos.expressions.algebra.`**`quantkeydist`**`(`**args*, ***kwargs*`)`

Shorthand for *`QuantumKeyDistribution`*.

`picos.expressions.algebra.`**`quantrelentr`**`(`**args*, ***kwargs*`)`

Shorthand for *`NegativeQuantumEntropy`*.

`picos.expressions.algebra.`**`rsoc`**`(`**args*, ***kwargs*`)`

Shorthand for *`RotatedSecondOrderCone`*.

`picos.expressions.algebra.`**`simplex`**`(`*gamma*`)`

Create a standard simplex of radius γ.

Deprecated since version 2.0: Use *`picos.Simplex`* instead.

`picos.expressions.algebra.`**`soc`**`(`**args*, ***kwargs*`)`

Shorthand for *`SecondOrderCone`*.

`picos.expressions.algebra.`**`sum`**`(`*lst*, *axis=None*`)`

Sum PICOS expressions and give the result a meaningful description.

This is a replacement for Python's *`sum`* that produces sensible string representations, and in some cases a speedup, when summing over multiple PICOS expressions. Additionally, this can be used to denote the (complete, column- or row-) sum over a single matrix expression.

Parameters

`axis` (*`None or int`*) – If summing over a single matrix expression, this is the axis over which the sum is performed: `None` denotes the sum over all elements, `0` denotes the sum of the rows as a row vector and 1 denotes the sum of the columns as a column vector. If summing multiple expressions, any value other than `None` raises a `ValueError`.

Example

```
>>> import builtins, picos, numpy
>>> x = picos.RealVariable("x", 5)
>>> e = [x[i]*x[i+1] for i in range(len(x) - 1)]
>>> builtins.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.sum(x)  # The same as x.sum or (1|x).
<1×1 Real Linear Expression: ∑(x)>
>>> A = picos.Constant("A", range(20), (4, 5))
>>> picos.sum(A, axis=0)  # The same as A.rowsum
<1×5 Real Constant: [1]ᵀ·A>
>>> picos.sum(A, axis=1)  # The same as A.colsum
<4×1 Real Constant: A·[1]>
>>> numpy.allclose(  # Same axis convention as NumPy.
...     numpy.sum(A.np, axis=0), picos.sum(A, axis=0).np)
True
```

`picos.expressions.algebra.`**`sum_k_largest`**(*x*, *k*)

Wrapper for *SumExtremes*.

Sets `largest = True` and `eigenvalues = False`.

Example

```
>>> from picos import RealVariable, sum_k_largest
>>> x = RealVariable("x", 5)
>>> sum_k_largest(x, 2)
<Sum of Largest Elements: sum_2_largest(x)>
>>> sum_k_largest(x, 2) <= 2
<Sum of Largest Elements Constraint: sum_2_largest(x) ≤ 2>
```

`picos.expressions.algebra.`**`sum_k_largest_lambda`**(*x*, *k*)

Wrapper for *SumExtremes*.

Sets `largest = True` and `eigenvalues = True`.

Example

```
>>> from picos import SymmetricVariable, sum_k_largest_lambda
>>> X = SymmetricVariable("X", 5)
>>> sum_k_largest_lambda(X, 2)
<Sum of Largest Eigenvalues: sum_2_largest_λ(X)>
>>> sum_k_largest_lambda(X, 2) <= 2
<Sum of Largest Eigenvalues Constraint: sum_2_largest_λ(X) ≤ 2>
```

`picos.expressions.algebra.`**`sum_k_smallest`**(*x*, *k*)

Wrapper for *SumExtremes*.

Sets `largest = False` and `eigenvalues = False`.

Example

```
>>> from picos import RealVariable, sum_k_smallest
>>> x = RealVariable("x", 5)
>>> sum_k_smallest(x, 2)
<Sum of Smallest Elements: sum_2_smallest(x)>
>>> sum_k_smallest(x, 2) >= 2
<Sum of Smallest Elements Constraint: sum_2_smallest(x) ≥ 2>
```

`picos.expressions.algebra.`**`sum_k_smallest_lambda`**(*x*, *k*)

Wrapper for *SumExtremes*.

Sets `largest = False` and `eigenvalues = True`.

Example

```
>>> from picos import SymmetricVariable, sum_k_smallest_lambda
>>> X = SymmetricVariable("X", 5)
>>> sum_k_smallest_lambda(X, 2)
<Sum of Smallest Eigenvalues: sum_2_smallest_λ(X)>
>>> sum_k_smallest_lambda(X, 2) >= 2
<Sum of Smallest Eigenvalues Constraint: sum_2_smallest_λ(X) ≥ 2>
```

`picos.expressions.algebra.`**`sumexp`**(**args*, ***kwargs*)

Shorthand for *SumExponentials*.

`picos.expressions.algebra.`**`trace`**(*x*)

Denote the trace of a square matrix.

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

Legacy shorthand for `PowerTrace`.

Deprecated since version 2.0: Use `picos.PowerTrace` instead.

`picos.expressions.algebra.`**`truncated_simplex`**(*gamma*, *sym=False*)

Create a truncated simplex of radius γ.

Deprecated since version 2.0: Use `picos.Simplex` instead.

Objects

`picos.expressions.algebra.`**`I`**

Create an identity matrix.

Example

```
>>> from picos import I
>>> print(I(3))
[ 1.00e+00     0          0    ]
[    0      1.00e+00     0    ]
[    0         0      1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c9e0>
```

`picos.expressions.algebra.`**`J`**

Create a matrix of all ones.

Example

```
>>> from picos import J
>>> print(J(2, 3))
[ 1.00e+00  1.00e+00  1.00e+00]
[ 1.00e+00  1.00e+00  1.00e+00]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655ca90>
```

`picos.expressions.algebra.`**`O`**

Create a zero matrix.

Example

```
>>> from picos import O
>>> print(O(2, 3))
[0 0 0]
[0 0 0]
```

Default value

```
<functools._lru_cache_wrapper object at 0x7d377655c930>
```

picos.expressions.cone

Backend for mathematical set type implementations.

Classes

class `picos.expressions.cone.`**`Cone`**(*dim*, *typeStr*, *symbStr*)

Bases: `Set`

Abstract base class for a cone.

`__init__`(*dim*, *typeStr*, *symbStr*)

Perform basic initialization for `Cone` instances.

Parameters

- **`dim`** (`int or None`) – Fixed member dimensionality, or `None`.
- **`typeStr`** (`str`) – Short string denoting the set type.
- **`symbStr`** (`str`) – Algebraic string description of the set.

`property dim`

The fixed member dimensionality, or `None`.

If this is `None`, the instance represents any finite dimensional version of the cone. Such an abstract cone can not be used to define a `ProductCone`.

`abstract property dual_cone`

The dual cone.

picos.expressions.cone_expcone

Implements `ExponentialCone`.

Classes

class `picos.expressions.cone_expcone.`**`ExponentialCone`**

Bases: `Cone`

The exponential cone.

Represents the convex cone $\operatorname{cl}\{(x, y, z) : y \exp(\frac{z}{y}) \leq x, x, y > 0\}$.

`__init__`()

Construct an exponential cone.

`property dual_cone`

Implement `cone.Cone.dual_cone`.

picos.expressions.cone_nno

Implements the nonnegative orthant cone.

Classes

`class picos.expressions.cone_nno.`**`NonnegativeOrthant`**`(`*dim=None*`)`

Bases: `Cone`

The nonnegative orthant.

`__init__``(`*dim=None*`)`

Construct a `NonnegativeOrthant`.

`property dual_cone`

Implement `cone.Cone.dual_cone`.

picos.expressions.cone_product

Implements a Cartesian product cone.

Classes

`class picos.expressions.cone_product.`**`ProductCone`**`(`**cones*`)`

Bases: `Cone`

A real Cartesian product cone.

`__init__``(`**cones*`)`

Construct a `ProductCone`.

Parameters

`cones` (`list(picos.expressions.Cone)`) – A sequence of cones to build the product cone from. May include other product cones that will be “unpacked” first.

`property cones`

The cones that make up the product cone as a tuple.

`property dual_cone`

Implement `cone.Cone.dual_cone`.

`property refined`

Overwrite `refined`.

picos.expressions.cone_psd

Implements the positive semidefinite cone.

Classes

class picos.expressions.cone_psd.**PositiveSemidefiniteCone**(*dim=None*)

Bases: `Cone`

The positive semidefinite cone.

Unlike other `cones` which are defined only on $\mathbb{K}^n$, this cone accepts both symmetric and hermitian matrices as well as their `special vectorization` as members.

Example

```
>>> from picos import Constant, PositiveSemidefiniteCone
>>> R = Constant("R", range(16), (4, 4))
>>> S = R + R.T
>>> S.shape
(4, 4)
>>> S.svec.shape
(10, 1)
>>> S.svec << PositiveSemidefiniteCone()  # Constrain the matrix via svec().
<4×4 LMI Constraint: R + Rᵀ ≽ 0>
>>> C = S << PositiveSemidefiniteCone(); C  # Constrain the matrix directly.
<4×4 LMI Constraint: R + Rᵀ ≽ 0>
>>> C.conic_membership_form[0]      # The conic form still refers to svec().
<10×1 Real Constant: svec(R + Rᵀ)>
>>> C.conic_membership_form[1]
<10-dim. Positive Semidefinite Cone: {svec(A) : xᵀ·A·x ≥ 0 ∀ x}>
```

__init__(*dim=None*)

Construct a `PositiveSemidefiniteCone`.

If a fixed dimensionality is given, this must be the dimensiona of the special vectorization. For a $n \times n$ matrix, this is $\frac{n(n+1)}{2}$.

property dual_cone

Implement `cone.Cone.dual_cone`.

picos.expressions.cone_rsoc

Implements `RotatedSecondOrderCone`.

Classes

class picos.expressions.cone_rsoc.**RotatedSecondOrderCone**(*p=1*, *dim=None*)

Bases: `Cone`

The (narrowed or widened) rotated second order cone.

For $n \in \mathbb{Z}_{\geq 3}$ and $p \in \mathbb{R}_{>0}$, represents the convex cone

$$\mathcal{R}_p^n = \left\{ x \in \mathbb{R}^n \;\middle|\; p x_1 x_2 \geq \sum_{i=3}^n x_i^2 \wedge x_1, x_2 \geq 0 \right\}.$$

For $p = 2$, this is the standard rotated second order cone $\mathcal{R}^n$ obtained by rotating the `second order cone` $\mathcal{Q}^n$ by $\frac{\pi}{4}$ in the (x_1, x_2) plane.

The default instance of this class has $p = 1$, which can be understood as a narrowed version of the standard cone. This is more convenient for defining the primal problem but it should be noted that $\mathcal{R}_1^n$ is not self-dual, so working with $p = 2$ may seem more natural when the dual problem is of interest.

Dual cone

The dual cone is

$$(\mathcal{R}_p^n)^* = \left\{ x \in \mathbb{R}^n \;\middle|\; \frac{4}{p} x_1 x_2 \geq \sum_{i=2}^{n} x_i^2 \wedge x_1, x_2 \geq 0 \right\} = \mathcal{R}_{4/p}^n.$$

The cone is thus self-dual for $p = 2$.

`__init__`(*p=1*, *dim=None*)

Construct a rotated second order cone.

Parameters

p (`float`) – The positive factor p in the definition.

`property dual_cone`

Implement `cone.Cone.dual_cone`.

`property p`

A narrowing ($p < 2$) or widening ($p > 2$) factor.

picos.expressions.cone_soc

Implements `SecondOrderCone`.

Classes

`class picos.expressions.cone_soc.`**`SecondOrderCone`**(*dim=None*)

Bases: `Cone`

The second order cone.

Also known as the quadratic, 2-norm, Lorentz, or ice cream cone.

For $n \in \mathbb{Z}_{\geq 2}$, represents the convex cone

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n \;\middle|\; x_1 \geq \sqrt{\sum_{i=2}^{n} x_i^2} \right\}.$$

Dual cone

The second order cone as defined above is self-dual.

`__init__`(*dim=None*)

Construct a second order cone.

`property dual_cone`

Implement `cone.Cone.dual_cone`.

picos.expressions.cone_trivial

Implements trivial cones.

Classes

class picos.expressions.cone_trivial.**TheField**(*dim=None*)

Bases: `Cone`

The real or complex field.

__init__(*dim=None*)

Construct a `TheField`.

property dual_cone

Implement `cone.Cone.dual_cone`.

class picos.expressions.cone_trivial.**ZeroSpace**(*dim=None*)

Bases: `Cone`

The set containing zero.

__init__(*dim=None*)

Construct a `ZeroSpace`.

property dual_cone

Implement `cone.Cone.dual_cone`.

picos.expressions.data

Functions to load and work with raw numeric data.

Functions

picos.expressions.data.**blend_shapes**(*baseShape*, *defaultShape*)

Replace wildcards in one shape with entries of the other.

Parameters

- **baseShape** (`tuple(int or None)`) – Primary shape, usually with wildcards.
- **defaultShape** (`tuple(int or None)`) – Secondary shape with fallback entries.

picos.expressions.data.**convert_and_refine_arguments**(**which*, *refine=True*, *allowNone=False*)

Convert selected function arguments to PICOS expressions.

If the selected arguments are already PICOS expressions, they are refined unless disabled. If they are not already PICOS expressions, an attempt is made to load them as constant expressions.

Decorator guarantee

All specified arguments are refined PICOS expressions when the function is exectued.

Parameters

- **refine** (`bool`) – Whether to refine arguments that are already PICOS expressions.
- **allowNone** (`bool`) – Whether `None` is passed through to the function.

picos.expressions.data.**convert_operands**(*sameShape=False*, *diagBroadcast=False*, *scalarLHS=False*, *scalarRHS=False*, *rMatMul=False*, *lMatMul=False*, *horiCat=False*, *vertCat=False*, *allowNone=False*)

Convert binary operator operands to PICOS expressions.

A decorator for a binary operator that converts any operand that is not already a PICOS expression or set into a constant one that fulfills the given shape requirements, if possible. See `load_data` for broadcasting and reshaping rules that apply to raw data.

If both operands are already PICOS expressions and at least one of them is an affine expression, there is a limited set of broadcasting rules to fix a detected shape mismatch. If this does not succeed, an exception is raised. If no operand is affine, the operation is performed even if shapes do not match. The operation is responsible for dealing with this case.

If either operand is a PICOS set, no broadcasting or reshaping is applied as set instances have, in general, variable dimensionality. If a set type can *not* have arbitrary dimensions, then it must validate the element's shape on its own. In particular, no shape requirement may be given when this decorator is used on a set method.

Decorator guarantee

This decorator guarantees to the binary operator using it that only PICOS expression or set types will be passed as operands and that any affine expression already has the proper shape for the operation, based on the decorator arguments.

Broadcasting rules for affine expressions

Currently, only scalar affine expressions are broadcasted to the next smallest matching shape. This is more limited than the broadcasting behavior when one of the operands is raw data, but it ensures a symmetric behavior in case both operands are affine. An exception is the case of diagBroadcast, where a vector affine expression may be extended to a matrix.

Parameters

- **`sameShape`** (*`bool`*) – Both operands must have the exact same shape.
- **`diagBroadcast`** (*`bool`*) – Both operands must be square matrices of same shape. If one operand is a square matrix and the other is a scalar or vector, the latter is put into the diagonal of a square matrix.
- **`scalarLHS`** (*`bool`*) – The left hand side operand must be scalar.
- **`scalarRHS`** (*`bool`*) – The right hand side operand must be scalar.
- **`rMatMul`** (*`bool`*) – The operation has the shape requirements of normal matrix multiplication with the second operand on the right side.
- **`lMatMul`** (*`bool`*) – The operation has the shape requirements of reversed matrix multiplication with the second operand on the left side.
- **`horiCat`** (*`bool`*) – The operation has the shape requirements of horizontal matrix concatenation.
- **`vertCat`** (*`bool`*) – The operation has the shape requirements of vertical matrix concatenation.
- **`allowNone`** (*`bool`*) – An operand of `None` is passed as-is.

Raises

`TypeError` – If matching shapes cannot be produced despite one of the operands being raw data or an affine expression.

Note: Matrix multiplication includes scalar by matrix multiplication, so either operand may remain a scalar.

`picos.expressions.data.`**`cvx2csc`**`(`*A*`)`

Convert a CVXOPT matrix to a SciPy sparse matrix in CSC format.

`picos.expressions.data.`**`cvx2csr`**`(`*A*`)`

Convert a CVXOPT matrix to a SciPy sparse matrix in CSR format.

`picos.expressions.data.`**`cvx2np`**`(`*A*, *reshape=None*`)`

Convert a CVXOPT (sparse) matrix to a NumPy two-dimensional array.

Parameters

- **A** – The CVXOPT `dense` or `sparse` matrix to convert.
- **reshape** (*bool*) – Optional new shape for the converted matrix.

Returns
Converted `NumPy array`.

`picos.expressions.data.`**`cvxopt_equals`**`(`*A*, *B*, *absTol=None*, *relTol=None*`)`

Whether two CVXOPT (sparse) matrices are numerically equal or close.

For every common entry of `A` and `B`, it is sufficient that one of the two tolerances, `absTol` or `relTol`, is satisfied.

Parameters

- **absTol** (*float*) – Maximum allowed entrywise absolute difference.
- **relTol** (*float*) – Maximum allowed entrywise quotient of absolute difference at the entry divided by the largest absolute value of any entry in both matrices.

`picos.expressions.data.`**`cvxopt_hcat`**`(`*matrices*`)`

Concatenate the given CVXOPT (sparse) matrices horizontally.

The resulting matrix is sparse if any input matrix is sparse.

Parameters
matrices (*list*) – A list of CVXOPT (sparse) matrices.

`picos.expressions.data.`**`cvxopt_hpd`**`(`*matrix*`)`

Whether the given CVXOPT matrix is hermitian positive definite.

Uses *RELATIVE_HERMITIANNESS_TOLERANCE*.

See also *cvxopt_hpsd*.

`picos.expressions.data.`**`cvxopt_hpsd`**`(`*matrix*`)`

Whether the given CVXOPT matrix is hermitian positive semidefinite.

Uses *RELATIVE_HERMITIANNESS_TOLERANCE* and *RELATIVE_SEMIDEFINITENESS_TOLERANCE*.

See also *cvxopt_hpd*.

Warning: The semidefiniteness tolerance allows negative, near-zero eigenvalues.

`picos.expressions.data.`**`cvxopt_inverse`**`(`*matrix*`)`

Return the inverse of the given CVXOPT matrix.

Raises
ValueError – If the matrix is not invertible.

`picos.expressions.data.`**`cvxopt_maxdiff`**`(`*A*, *B*`)`

Return the largest absolute difference of two (sparse) CVXOPT matrices.

Raises
TypeError – If the matrices are not of the same shape.

`picos.expressions.data.`**`cvxopt_principal_root`**`(`*matrix*`)`

Return the principal square root of a symmetric positive semidef. matrix.

Given a real symmetric positive (semi)definite CVXOPT input matrix, returns its unique positive (semi)definite matrix square root.

Warning: Does not validate that the input matrix is symmetric positive semidefinite and will still return a (useless) matrix if it is not.

`picos.expressions.data.`**`cvxopt_vcat`**`(`*matrices*`)`

Concatenate the given CVXOPT (sparse) matrices vertically.

The resulting matrix is sparse if any input matrix is sparse.

Parameters

`matrices` (*`list`*) – A list of CVXOPT (sparse) matrices.

`picos.expressions.data.`**`is_scipy_spmat`**`(`*value*`)`

Report whether value is a SciPy sparse matrix without importing scipy.

`picos.expressions.data.`**`left_kronecker_I`**`(`*A*, *k*, *reshape=None*, *preT=False*, *postT=False*`)`

Return $I_k \otimes A$ for a CVXOPT (sparse) matrix A.

In other words, if A is a $m \times n$ CVXOPT (sparse) matrix, returns a $km \times kn$ CVXOPT sparse block matrix with all blocks of size $m \times n$, the diagonal blocks (horizontal block index equal to vertical block index) equal to A, and all other blocks zero.

Parameters

- **`reshape`** – If set, then A is reshaped on the fly.
- **`preT`** (*`bool`*) – Transpose A before reshaping.
- **`postT`** (*`bool`*) – Transpose A after reshaping.

Returns

If A is dense and $k = 1$, a `CVXOPT dense matrix`, otherwise a `CVXOPT sparse matrix`.

`picos.expressions.data.`**`load_data`**`(`*value*, *shape=None*, *typecode=None*, *sparse=None*, *alwaysCopy=True*, *legacy=False*`)`

Load a constant numeric data value as a CVXOPT (sparse) matrix.

As a user, you never need to call this manually, but you should be aware that PICOS uses this function on any raw data you supply as an operand when working with PICOS expressions. For instance, you can just add a NumPy matrix or an algebraic string such as `"I"` to such an expression without worrying about any conversion.

Supported data types

- A NumPy matrix: `numpy.ndarray` or `numpy.matrix`.
- A SciPy sparse matrix: All from `scipy.sparse`.
- A CVXOPT matrix: `cvxopt.matrix` or `cvxopt.spmatrix`.
- A constant PICOS expression: *`AffineExpression`* or *`ComplexAffineExpression`*.
- A Python scalar: `int`, `float` or `complex`.
- A flat `tuple` or `list` containing scalars or a `range`, all representing a column vector.
- A nested `tuple` or `list` containing scalars. The outer level represents rows and the inner level represents the rows' entries. Allows you to define a 2×3 matrix like this:

```
A = [[1, 2, 3],
     [4, 5, 6]]
```

- A verbatim string description, with rows separated by newline and columns separated by whitespace. The same 2×3 matrix as above:

```
A = '''1 2 3
       4 5 6'''
```

- An algebraic string description:

`"|a|"`	A matrix with all entries equal to a.
`"|a|(m,n)"`	A $m \times n$ matrix with all entries equal to a.
`"e_i,j(m,n)"`	A $m \times n$ matrix with a 1 at (i, j), indexed from $(1, 1)$ to (m, n).
`"e_i(m,n)"`	A $m \times n$ matrix with a single 1 on the i-th coordinate, indexed from 1 in column-major order.
`"I"`	The identity matrix.
`"I(n)"`	The $n \times n$ identiy matrix.
`"a..."`	The matrix given by `"..."` but multiplied by a.

Different matrix operations such as addition or multiplication have different requirements with respect to the operands' shapes. The shape of any PICOS expression involved will always be maintained. But when an operation involves both a PICOS expression and raw data, then PICOS will try to broadcast or reshape the raw data such that the operation can be performed.

Broadcasting and reshaping rules

- An input vector without a second axis (for instance a non-nested `tuple` or `list` or a `range`) is interpreted as a row vector if the target shape is of the form `(None, n)` with $n > 1$, otherwise it is interpreted as a column vector.
- If the target shape is `None` or `(None, None)`, then the input's shape is maintained.
- If the target shape contains `None` exactly once, that occurence is replaced by the respective dimensionality of the input data shape.
- A scalar is copied (broadcasted) to fill the target shape.
- A column (row) vector is copied horizontally (vertically) if its length matches the target row (column) count.
- Reshaping from matrix to vector: A matrix is vectorized in column-major (row-major) order if the target shape is a column (row) vector whose length matches the number of matrix entries.
- Reshaping from vector to matrix: A column (row) vector is treated as the column-major (row-major) vectorization of the target matrix if the former's length matches the number of the latter's entries.
- All other combinations of input and target shape raise an exception.
- When an algebraic string description specifies no shape, then the shape argument must be supplied. When both the string and the shape argument specify a shape, then they must be consistent (no broadcasting or reshaping is applied in this case).

Parameters

- **shape** (*int or tuple or list or None*) – The shape of the resulting matrix. If the input data is of another shape, broadcasting or reshaping is used if possible, otherwise an exception is raised. An integer is treated as the length of a column vector. If this is `None`, then the target's shape is the input's. If only the target number of rows or columns is `None`, then only that quantity is chosen according to the input.
- **typecode** (*str*) – The numeric type of the resulting matrix. Either `'d'` (float), `'z'` (complex) or `'i'` (integer). If the input data is not already of this type, then it will be converted if possible. If this is not possible, then an exception is raised. If this is `None`, then the output type is chosen based on the input data.
- **sparse** (*bool or None*) – If `True`, a sparse matrix is returned. If `False`, a dense matrix is returned. If `None`, it depends on the sparsity pattern of the input data. If the

typecode argument is `'i'`, then a value of `True` is not allowed and the returned matrix is dense.

- **`alwaysCopy`** (*bool*) – If `True`, then a copy of the input data is returned even if it already equals the output data. If `False`, the input value can be returned as such if it is already a CVXOPT matrix of the desired shape, typecode, and sparsity.
- **`legacy`** (*bool*) – Be compatible with the old `retrieve_matrix` function. In particular, if the target shape contains `None` exactly once and the input data is scalar, treat this as a matrix multiplication case and return the scalar times an identity matrix of appropriate size.

Returns

A `tuple` whose first entry is the loaded matrix and whose second argument is a short string for representing the data within algebraic expression strings.

Example

```
>>> from picos.expressions.data import load_data
>>> # Data as (nested) list:
>>> load_data([1,2,3])
(<3x1 matrix, tc='i'>, '[3×1]')
>>> load_data([[1,2,3]])
(<1x3 matrix, tc='i'>, '[1×3]')
>>> A = [[1,2,3],
...      [4,5,6]]
>>> load_data(A)
(<2x3 matrix, tc='i'>, '[2×3]')
>>> # Data as string:
>>> value, string = load_data('e_14(7,2)')
>>> print(string)
[7×2:e_7,2]
>>> print(value)
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        0       ]
[   0        1.00e+00]
>>> load_data('5.3I', (2,2))
(<2x2 sparse matrix, tc='d', nnz=2>, '5.3·I')
```

`picos.expressions.data.`**`load_dense_data`**(*value*, *shape=None*, *typecode=None*, *alwaysCopy=True*)

See *`load_data`* with `sparse = False`.

`picos.expressions.data.`**`load_shape`**(*shape*, *squareMatrix=False*, *wildcards=False*)

Parse the argument as a matrix shape.

PICOS uses this function whenever you supply a shape parameter to a method.

A scalar argument is treated as the length of a column-vector. If the shape contains `None`, it is treated as a wildcard (any dimensionality).

Parameters

- **`squareMatrix`** (*bool*) – If `True`, a scalar argument is treated as the side/diagonal length of a square matrix, and any other argument is validated to be square. If `False`, a scalar argument is treated as the length of a column vector.
- **`wildcards`** (*bool*) – Whether the wildcard token `None` is allowed.

`picos.expressions.data.`**`load_sparse_data`**(*value*, *shape=None*, *typecode=None*, *alwaysCopy=True*)

See *`load_data`* with `sparse = True`.

`picos.expressions.data.`**`make_fraction`**(*p*, *denominator_limit*)

Convert a float p to a limited precision fraction.

Parameters

- **`p`** (*`float`*) – The float to convert, may be positive or negative infinity.
- **`denominator_limit`** (*`int`*) – The largest allowed denominator.

Returns tuple

A quadruple `(num, den, pNew, pStr)` with `pNew` the limited precision version of p, `pStr` a string representation of the fraction, and `num` and `den` the numerator and the denominator of the fraction, respectively.

`picos.expressions.data.`**`right_kronecker_I`**(*A*, *k*, *reshape=None*, *preT=False*, *postT=False*)

Return $A \otimes I_k$ for a CVXOPT (sparse) matrix A.

Parameters

- **`reshape`** – If set, then A is reshaped on the fly.
- **`preT`** (*`bool`*) – Transpose A before reshaping.
- **`postT`** (*`bool`*) – Transpose A after reshaping.

Returns

If A is dense and $k = 1$, a `CVXOPT dense matrix`, otherwise a `CVXOPT sparse matrix`.

`picos.expressions.data.`**`should_be_sparse`**(*shape*, *numNonZero*)

Decide whether a matrix is considered sparse.

Parameters

- **`shape`** (*`tuple(int)`*) – The shape of the matrix in question.
- **`numNonZero`** (*`int`*) – Number of non-zero elements of the matrix.

`picos.expressions.data.`**`sp2cvx`**(*A*)

Convert a SciPy sparse matrix to a CVXOPT sparse matrix.

`picos.expressions.data.`**`sparse_quadruple`**(*A*, *reshape=None*, *preT=False*, *postT=False*)

Return a sparse representation of the given CVXOPT (sparse) matrix.

Parameters

- **`reshape`** – If set, then A is reshaped on the fly.
- **`preT`** (*`bool`*) – Transpose A before reshaping.
- **`postT`** (*`bool`*) – Transpose A after reshaping.

Returns

A quadruple of values, row indices, column indices, and shape.

`picos.expressions.data.`**`value`**(*obj*, *sparse=None*, *numpy=False*)

Convert (nested) PICOS objects to their current value.

Parameters

- **`obj`** – Either a single (PICOS) object that has a `value` attribute, such as a *`mutable`*, *`expression`* or *`Problem`*, or a (nested) `list`, `tuple` or `dict` thereof.
- **`sparse`** – If `None`, retrieved multidimensional values can be returned as either CVX-OPT `sparse` or `dense` matrices, whichever PICOS stores internally. If `True` or `False`, multidimensional values are always returned as sparse or dense types, respectively.

- **numpy** (*bool*) – If True, retrieved multidimensional values are returned as a NumPy ndarray instead of a CVXOPT type. May not be set in combination with sparse=True.

Returns

An object of the same (nested) structure as obj, with every occurence of any object with a value attribute replaced by that attribute's current numeric value. In the case of dictionaries, only the dictionary values will be converted.

Raises

TypeError – If some object with a value attribute has a value that cannot be converted to a matrix by *load_data*. This can only happen if the object in question is not a PICOS object.

Example

```
>>> from picos import RealVariable, value
>>> from pprint import pprint
>>> x = {key: RealVariable(key) for key in ("foo", "bar")}
>>> x["foo"].value = 2
>>> x["bar"].value = 3
>>> pprint(value(x))
{'bar': 3.0, 'foo': 2.0}
```

Objects

picos.expressions.data.**TOLERANCE**

Maximum entrywise absolute deviation allowed for numeric equality checks.

Default value

```
1e-06
```

picos.expressions.data.**cvxopt_K**

The commutation matrix $K_{(m,n)}$ as a CVXOPT sparse matrix.

Default value

```
<functools._lru_cache_wrapper object at 0x7d37769792d0>
```

picos.expressions.exp_affine

Implements affine expression types.

Classes

class picos.expressions.exp_affine.**AffineExpression**(*string*, *shape=(1, 1)*, *coefficients={}*)

Bases: *ComplexAffineExpression*

A multidimensional real affine expression.

__eq__(*other*)

Return an equality constraint concerning the expression.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

property H

The regular transpose of the `AffineExpression`.

property conj

The `AffineExpression` as is.

property exp

The exponential function applied to the expression.

property imag

A zero of same shape as the `AffineExpression`.

property isreal

Always true for `AffineExpression` instances.

property log

The Logarithm of the expression.

property real

The `AffineExpression` as is.

class picos.expressions.exp_affine.ComplexAffineExpression(*string*, *shape=(1, 1)*, *coefficients={}*)

Bases: `BiaffineExpression`

A multidimensional (complex) affine expression.

Base class for the real `AffineExpression`.

__abs__()

Denote the default norm of the expression.

The norm used depends on the expression's domain. It is

1. the absolute value of a real scalar,
2. the modulus of a complex scalar,
3. the Euclidean norm of a vector, and
4. the Frobenius norm of a matrix.

__eq__(*other*)

Return an equality constraint concerning the expression.

__mul__(*other*)

Denote multiplication with another expression on the right.

__or__(*other*)

Denote the scalar product with another expression on the right.

For (complex) vectors a and b this is the dot product

$$\begin{aligned}(a \mid b) &= \langle a \mid b \rangle \\ &= \langle a, b \rangle \\ &= a \cdot b \\ &= a^H b.\end{aligned}$$

For (complex) matrices A and B this is the Frobenius inner product

$$\begin{aligned}(A \mid B) &= \langle A, B \rangle_F \\ &= A : B \\ &= \operatorname{tr}(A^H B) \\ &= \operatorname{vec}(\overline{A})^T \operatorname{vec}(B)\end{aligned}$$

Note: Write `(A|B)` instead of `A|B` for the scalar product of `A` and `B` to obtain correct operator binding within a larger expression context.

`__xor__`(*other*)

Denote the entrywise product with another expression on the right.

`scipy_sparse_matrix_form`(*varOffsetMap*, ***, *offset=0*, *padding=0*, *dense_b=False*)

Like *`sparse_matrix_form`* but returns SciPy types.

See *`sparse_matrix_form`* for details and arguments.

Returns tuple(scipy.sparse.csc_matrix)
A pair `(A, b)` of SciPy sparse matrices in CSC format representing the matrix A and the column vector b. (If `dense_b=True`, then `b` is returned as a 1-D `NumPy array` instead.)

Raises
`ModuleNotFoundError` – If the optional dependency `scipy` is not installed.

`sparse_matrix_form`(*varOffsetMap*, ***, *offset=0*, *padding=0*, *dense_b=False*)

Return a representation suited for embedding in constraint matrices.

This computes a sparse matrix A and a sparse column vector b such that $Ax+b$ represents the vectorized expression, where x is a vertical concatenation of a number of variables, including those that appear in the expression. The size and ordering of x is given through `varOffsetMap`, which maps PICOS variables to their starting position within x.

If the optional parameters `offset` and `padding` are given, then both A and b are padded with zero rows from above and below, respectively.

This method is used by PICOS internally to assemble constraint matrices.

Parameters

- **`varOffsetMap`** (*`dict`*) – Maps variables to column offsets.
- **`offset`** (*`int`*) – Number of zero rows to insert at the top of A and b.
- **`offset`** – Number of zero rows to insert at the bottom of A and b.
- **`dense_b`** (*`bool`*) – Whether to return b as a dense vector. Not compatible with nonzero `offset` or `padding`.

Returns tuple(cvxopt.spmatrix)
A pair `(A, b)` of CVXOPT sparse matrices representing the matrix A and the column vector b. (If `dense_b=True`, then `b` is returned as a dense CVXOPT column vector instead.)

`sparse_rows`(*varOffsetMap*)

Yield a sparse list representation of the expression.

This is similar to *`sparse_matrix_form`* (with default arguments) but instead of returning A and b at once, this yields for every row of $[A \mid b]$, each representing a scalar entry of the expression's vectorization, a triplet containing a list of column indices and values of that row of A and the entry of b.

Parameters
`varOffsetMap` – Maps variables to column offsets.

Yields tuple(list, list, float)
Triples `(J, V, c)` where `J` contains column indices (representing scalar variables), `V` contains coefficients for each column index, and where `c` is a constant term.

`property imag`

Override *`imag`*.

The result is returned as an *`AffineExpression`*.

property real

Override *real*.

The result is returned as an *AffineExpression*.

Functions

picos.expressions.exp_affine.**Constant**(*name_or_value*, *value=None*, *shape=None*)

Create a constant PICOS expression.

Loads the given numeric value as a constant *ComplexAffineExpression* or *AffineExpression*, depending on the value. Optionally, the value is broadcasted or reshaped according to the shape argument.

Parameters

- **name_or_value** (*str*) – Symbolic string description of the constant. If None or the empty string, a string will be generated. If this is the only positional parameter (i.e.``value`` is not given), then this position is used as the value argument instead!
- **value** – The numeric constant to load.

See *load_data* for supported data formats and broadcasting and reshaping rules.

Example

```
>>> from picos import Constant
>>> Constant(1)
<1×1 Real Constant: 1>
>>> Constant(1, shape=(2, 2))
<2×2 Real Constant: [1]>
>>> Constant("one", 1)
<1×1 Real Constant: one>
>>> Constant("J", 1, (2, 2))
<2×2 Real Constant: J>
```

picos.expressions.exp_biaffine

Implements *BiaffineExpression*.

Classes

class picos.expressions.exp_biaffine.**BiaffineExpression**(*string*, *shape=(1, 1)*, *coefficients={}*)

Bases: *Expression*, ABC

A multidimensional (complex) biaffine expression.

Abstract base class for the affine *ComplexAffineExpression* and its real subclass *AffineExpression*. Quadratic expressions are stored in *QuadraticExpression* instead.

In general this expression has the form

$$A(x, y) = B(x, y) + P(x) + Q(y) + C$$

where $x \in \mathbb{R}^p$ and $y \in \mathbb{R}^q$ are variable vectors, $B : \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{C}^{m \times n}$ is a bilinear function, $P : \mathbb{R}^p \to \mathbb{C}^{m \times n}$ and $Q : \mathbb{R}^q \to \mathbb{C}^{m \times n}$ are linear functions, and $C \in \mathbb{C}^{m \times n}$ is a constant.

If no coefficient matrices defining B and Q are provided on subclass instanciation, then this acts as an affine function of x.

In a more technical sense, the notational variables x and y each represent a stack of vectorizations of a number of *actual* scalar, vector, or matrix variables or parameters X_i and Y_j with $i, j \in \mathbb{Z}_{\geq 0}$ and this class stores the nonzero (bi)linear functions $B_{i,j}(\operatorname{vec}(X_i), \operatorname{vec}(Y_j))$, $P_i(\operatorname{vec}(X_i))$ and $Q_j(\operatorname{vec}(Y_j))$ in the form of separate sparse coefficient matrices.

`Htranspose``()`

Return the conjugate (or Hermitian) transpose.

Deprecated since version 2.0: Use `H` instead.

`__add__``(`*other*`)`

Denote addition with another expression on the right-hand side.

`__and__``(`*other*`)`

Denote horizontal stacking with another expression on the right.

`__floordiv__``(`*other*`)`

Denote vertical stacking with another expression below.

`__init__``(`*string*, *shape=(1, 1)*, *coefficients={}*`)`

Initialize a (complex) biaffine expression.

Parameters

- **`string`** (`str`) – A symbolic string description.
- **`shape`** (`int or tuple or list`) – Shape of a vector or matrix expression.
- **`coefficients`** (`dict`) – Maps mutable pairs, single mutables or the empty tuple to sparse represetations of the coefficient matrices B, P and Q, and C, respectively.

Warning: If the given coefficients are already of the desired (numeric) type and shape, they are stored by reference. Modifying such data can lead to unexpected results as PICOS expressions are supposed to be immutable (to allow caching of results).

If you create biaffine expressions by any other means than this constructor, PICOS makes a copy of your data to prevent future modifications to it from causing inconsistencies.

`__matmul__``(`*other*`)`

Denote the Kronecker product with another expression on the right.

`__mul__``(`*other*`)`

Denote multiplication with another expression on the right.

`__neg__``()`

Denote the negation of the expression.

`__or__``(`*other*`)`

Denote the scalar product with another expression on the right.

For (complex) vectors a and b this is the dot product

$$\begin{aligned}(a \mid b) &= \langle a \mid b \rangle \\ &= \langle a, b \rangle \\ &= a \cdot b \\ &= a^H b.\end{aligned}$$

For (complex) matrices A and B this is the Frobenius inner product

$$\begin{aligned}(A \mid B) &= \langle A, B \rangle_F \\ &= A : B \\ &= \operatorname{tr}(A^H B) \\ &= \operatorname{vec}(\overline{A})^T \operatorname{vec}(B)\end{aligned}$$

Note: Write `(A|B)` instead of `A|B` for the scalar product of `A` and `B` to obtain correct operator binding within a larger expression context.

`__pow__`(*other*)
Denote exponentiation with another, scalar expression.

`__radd__`(*other*)
Denote addition with another expression on the left-hand side.

`__rand__`(*other*)
Denote horizontal stacking with another expression on the left.

`__rfloordiv__`(*other*)
Denote vertical stacking with another expression above.

`__rmatmul__`(*other*)
Denote the Kronecker product with another expression on the left.

`__rmul__`(*other*)
Denote multiplication with another expression on the left.

`__ror__`(*other*)
Denote the scalar product with another expression on the left.

See `__or__` for details on this operation.

`__rsub__`(*other*)
Denote subtraction of the expression from another expression.

`__rtruediv__`(*other*)
Denote scalar division of another expression.

`__rxor__`(*other*)
Denote the entrywise product with another expression on the left.

`__sub__`(*other*)
Denote subtraction of another expression from the expression.

`__truediv__`(*other*)
Denote division by another, scalar expression.

`__xor__`(*other*)
Denote the entrywise product with another expression on the right.

`broadcasted`(*shape*)
Return the expression broadcasted to the given shape.

Example

```
>>> from picos import Constant
>>> C = Constant("C", range(6), (2, 3))
>>> print(C)
[ 0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00]
>>> print(C.broadcasted((6, 6)))
[ 0.00e+00  2.00e+00  4.00e+00  0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00  1.00e+00  3.00e+00  5.00e+00]
[ 0.00e+00  2.00e+00  4.00e+00  0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00  1.00e+00  3.00e+00  5.00e+00]
[ 0.00e+00  2.00e+00  4.00e+00  0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00  1.00e+00  3.00e+00  5.00e+00]
```

conjugate()

Return the complex conjugate.

Deprecated since version 2.0: Use `conj` instead.

copy()

Return a deep copy of the expression.

Deprecated since version 2.0: PICOS expressions are now immutable.

dupdiag(*n*)

Return a matrix with the (repeated) expression on the diagonal.

Vectorization is performed in column-major order.

Parameters
n (*int*) – Number of times to duplicate the vectorization.

dupvec(*n*)

Return a (repeated) column-major vectorization of the expression.

Parameters
n (*int*) – Number of times to duplicate the vectorization.

Returns
A column vector.

Example

```
>>> from picos import Constant
>>> A = Constant("A", [[1, 2], [3, 4]])
>>> A.dupvec(1) is A.vec
True
>>> A.dupvec(3).equals(A.vec // A.vec // A.vec)
True
```

equals(*other*, *absTol=None*, *relTol=None*)

Check mathematical equality with another (bi)affine expression.

The precise type of both (bi)affine expressions may differ. In particular, a *ComplexAffineExpression* with real coefficients and constant term can be equal to an *AffineExpression*.

If the operand is not already a PICOS expression, an attempt is made to load it as a constant affine expression. In this case, no reshaping or broadcasting is used to bring the constant term to the same shape as this expression. In particular,

- `0` refers to a scalar zero (see also *is0*),
- lists and tuples are treated as column vectors and
- algebraic strings must specify a shape (see *load_data*).

Parameters

- **other** – Another PICOS expression or a constant numeric data value supported by *load_data*.
- **absTol** – As long as all absolute differences between scalar entries of the coefficient matrices and the constant terms being compared does not exceed this bound, consider the expressions equal.
- **relTol** – As long as all absolute differences between scalar entries of the coefficient matrices and the constant terms being compared divided by the maximum absolute value found in either term does not exceed this bound, consider the expressions equal.

Example

```
>>> from picos import Constant
>>> A = Constant("A", 0, (5,5))
>>> repr(A)
'<5×5 Real Constant: A>'
>>> A.is0
True
>>> A.equals(0)
False
>>> A.equals("|0|(5,5)")
True
>>> repr(A*1j)
'<5×5 Complex Constant: A·1j>'
>>> A.equals(A*1j)
True
```

factor_out(*mutable*)

Factor out a single mutable from a vector expression.

If this expression is a column vector a that depends on some mutable x with a trivial internal vectorization format (i.e. *FullVectorization*), then this method, called with x as its argument, returns a pair of expressions (a_x, a_0) such that $a = a_x \operatorname{vec}(x) + a_0$.

Returns

Two refined *BiaffineExpression* instances that do not depend on `mutable`.

Raises

- **TypeError** – If the expression is not a column vector or if `mutable` is not a *Mutable* or does not have a trivial vectorization format.
- **LookupError** – If the expression does not depend on `mutable`.

Example

```
>>> from picos import RealVariable
>>> from picos.uncertain import UnitBallPerturbationSet
>>> x = RealVariable("x", 3)
>>> z = UnitBallPerturbationSet("z", 3).parameter
>>> a = ((2*x + 3)^z) + 4*x + 5; a
<3×1 Uncertain Affine Expression: (2·x + [3])⊙z + 4·x + [5]>
>>> sorted(a.mutables, key=lambda mtb: mtb.name)
[<3×1 Real Variable: x>, <3×1 Perturbation: z>]
>>> az, a0 = a.factor_out(z)
>>> az
<3×3 Real Affine Expression: ((2·x + [3])⊙z + 4·x + [5])_z>
>>> a0
<3×1 Real Affine Expression: ((2·x + [3])⊙z + 4·x + [5])_0>
>>> sorted(az.mutables.union(a0.mutables), key=lambda mtb: mtb.name)
[<3×1 Real Variable: x>]
>>> (az*z + a0).equals(a)
True
```

classmethod fromMatrix(*matrix*, *size=None*)

Create a class instance from a numeric matrix.

Deprecated since version 2.0: Use *from_constant* instead.

classmethod fromScalar(*scalar*)

Create a class instance from a numeric scalar.

Deprecated since version 2.0: Use *from_constant* instead.

classmethod from_constant(*constant*, *shape=None*, *name=None*)

Create a class instance from the given numeric constant.

Loads the given constant as a PICOS expression, optionally broadcasted or reshaped to the given shape and named as specified.

See `load_data` for supported data formats and broadcasting and reshaping rules.

Unlike `Constant`, this class method always creates an instance of the class that it is called on, instead of tailoring towards the numeric type of the data.

Note: When an operation involves both a PICOS expression and a constant value of another type, PICOS converts the constant on the fly so that you rarely need to use this method.

hadamard(*fact*)

Denote the elementwise (or Hadamard) product.

Deprecated since version 2.0: Use `object.__xor__` instead.

isconstant()

Whether the expression involves no mutables.

Deprecated since version 2.0: Use `constant` instead.

kron(*other*)

Denote the Kronecker product with another expression on the right.

Deprecated since version 2.2: Use infix @ instead.

leftkron(*other*)

Denote the Kronecker product with another expression on the left.

Deprecated since version 2.2: Reverse operands and use infix @ instead.

partial_trace(*subsystems*, *dimensions=2*)

Return the partial trace over selected subsystems.

If the expression can be written as $A_0 \otimes \cdots \otimes A_{n-1}$ for matrices $A_0, \ldots, A_{n-1}$ with shapes given in `dimensions`, then this returns $B_0 \otimes \cdots \otimes B_{n-1}$ with $B_i = \operatorname{tr}(A_i)$, if `i in subsystems` (with $i = -1$ read as $n - 1$), and $B_i = A_i$, otherwise.

Parameters

- **subsystems** (*int or tuple or list*) – A collection of or a single subystem number, indexed from zero, corresponding to subsystems that shall be traced over. The value -1 refers to the last subsystem.
- **dimensions** (*int or tuple or list*) – Either an integer d so that the subsystems are assumed to be all of shape $d \times d$, or a sequence of subsystem shapes where an integer d within the sequence is read as $d \times d$. In any case, the elementwise product over all subsystem shapes must equal the expression's shape.

Raises

- **TypeError** – If the subsystems do not match the expression or if a non-square subsystem is to be traced over.
- **IndexError** – If the subsystem selection is invalid in any other way.

Example

```
>>> from picos import Constant
>>> A = Constant("A", range(16), (4, 4))
>>> print(A)
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
```

(continues on next page)

(continued from previous page)

```
[ 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]
>>> A0 = A.partial_trace(0); A0
<2×2 Real Constant: A.{tr([2×2])⊗[2×2]}>
>>> print(A0)
[ 1.00e+01  1.80e+01]
[ 1.20e+01  2.00e+01]
>>> A1 = A.partial_trace(1); A1
<2×2 Real Constant: A.{[2×2]⊗tr([2×2])}>
>>> print(A1)
[ 5.00e+00  2.10e+01]
[ 9.00e+00  2.50e+01]
```

partial_transpose(*subsystems*, *dimensions=2*)

Return the expression with selected subsystems transposed.

If the expression can be written as $A_0 \otimes \cdots \otimes A_{n-1}$ for matrices $A_0, \ldots, A_{n-1}$ with shapes given in `dimensions`, then this returns $B_0 \otimes \cdots \otimes B_{n-1}$ with $B_i = A_i^T$, if `i in subsystems` (with $i = -1$ read as $n - 1$), and $B_i = A_i$, otherwise.

Parameters

- **subsystems** (*int or tuple or list*) – A collection of or a single subystem number, indexed from zero, corresponding to subsystems that shall be transposed. The value -1 refers to the last subsystem.
- **dimensions** (*int or tuple or list*) – Either an integer d so that the subsystems are assumed to be all of shape $d \times d$, or a sequence of subsystem shapes where an integer d within the sequence is read as $d \times d$. In any case, the elementwise product over all subsystem shapes must equal the expression's shape.

Raises

- `TypeError` – If the subsystems do not match the expression.
- `IndexError` – If the subsystem selection is invalid.

Example

```
>>> from picos import Constant
>>> A = Constant("A", range(16), (4, 4))
>>> print(A)
[ 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]
>>> A0 = A.partial_transpose(0); A0
<4×4 Real Constant: A.{[2×2]ᵀ⊗[2×2]}>
>>> print(A0)
[ 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]
>>> A1 = A.partial_transpose(1); A1
<4×4 Real Constant: A.{[2×2]⊗[2×2]ᵀ}>
>>> print(A1)
[ 0.00e+00  1.00e+00  8.00e+00  9.00e+00]
[ 4.00e+00  5.00e+00  1.20e+01  1.30e+01]
```

(continues on next page)

(continued from previous page)

```
[ 2.00e+00  3.00e+00  1.00e+01  1.10e+01]
[ 6.00e+00  7.00e+00  1.40e+01  1.50e+01]
```

renamed(*string*)

Return the expression with a modified string description.

reshaped(*shape*, *order='F'*)

Return the expression reshaped in the given order.

The default indexing order is column-major. Given an $m \times n$ matrix, reshaping in the default order is a constant time operation while reshaping in row-major order requires $O(mn)$ time. However, the latter allows you to be consistent with NumPy, which uses C-order (a generalization of row-major) by default.

Parameters

order (*str*) – The indexing order to use when reshaping. Must be either `"F"` for Fortran-order (column-major) or `"C"` for C-order (row-major).

Example

```
>>> from picos import Constant
>>> C = Constant("C", range(6), (2, 3))
>>> print(C)
[ 0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00]
>>> print(C.reshaped((3, 2)))
[ 0.00e+00  3.00e+00]
[ 1.00e+00  4.00e+00]
[ 2.00e+00  5.00e+00]
>>> print(C.reshaped((3, 2), order="C"))
[ 0.00e+00  2.00e+00]
[ 4.00e+00  1.00e+00]
[ 3.00e+00  5.00e+00]
```

reshaped_or_broadcasted(*shape*)

Return the expression `reshaped` or `broadcasted`.

Unlike with `reshaped` and `broadcasted`, the target shape may not contain a wildcard character.

If the wildcard-free target shape has the same number of elements as the current shape, then this is the same as `reshaped`, otherwise it is the same as `broadcasted`.

reshuffled(*permutation='ikjl'*, *dimensions=None*, *order='C'*)

Return the reshuffled or realigned expression.

This operation works directly on matrices. However, it is equivalent to the following sequence of operations:

1. The matrix is reshaped to a tensor with the given `dimensions` and according to `order`.
2. The tensor's axes are permuted according to `permutation`.
3. The tensor is reshaped back to the shape of the original matrix according to `order`.

For comparison, the following function applies the same operation to a 2D NumPy `ndarray`:

```
def reshuffle_numpy(matrix, permutation, dimensions, order):
    P = "{} -> {}".format("".join(sorted(permutation)), permutation)
    reshuffled = numpy.reshape(matrix, dimensions, order)
    reshuffled = numpy.einsum(P, reshuffled)
    return numpy.reshape(reshuffled, matrix.shape, order)
```

Parameters

- **permutation** (*str or tuple or list*) – A sequence of comparable elements with length equal to the number of tensor dimensions. The sequence is compared to its ordered version and the resulting permutation pattern is used to permute the tensor indices. For instance, the string "ikjl" is compared to its sorted version "ijkl" and denotes that the second and third axis should be swapped.
- **dimensions** (*None or tuple or list*) – If this is an integer sequence, then it defines the dimensions of the tensor. If this is None, then the tensor is assumed to be hypercubic and the number of dimensions is inferred from the permutation argument.
- **order** (*str*) – The indexing order to use for the virtual reshaping. Must be either "F" for Fortran-order (generalization of column-major) or "C" for C-order (generalization of row-major). Note that PICOS usually reshapes in Fortran-order while NumPy defaults to C-order.

Example

```
>>> from picos import Constant
>>> A = Constant("A", range(16), (4, 4))
>>> print(A)
[ 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]
>>> R = A.reshuffled(); R
<4×4 Real Constant: shuffled(A,ikjl,C)>
>>> print(R)
[ 0.00e+00  4.00e+00  1.00e+00  5.00e+00]
[ 8.00e+00  1.20e+01  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  3.00e+00  7.00e+00]
[ 1.00e+01  1.40e+01  1.10e+01  1.50e+01]
>>> A.reshuffled("ji").equals(A.T)     # Regular transposition.
True
>>> A.reshuffled("3214").equals(A.T0)  # Partial transposition (1).
True
>>> A.reshuffled("1432").equals(A.T1)  # Partial transposition (2).
True
```

same_as(*other*)

Check mathematical equality with another affine expression.

Deprecated since version 2.0: Use *equals* instead.

soft_copy()

Return a shallow copy of the expression.

Deprecated since version 2.0: PICOS expressions are now immutable.

transpose()

Return the matrix transpose.

Deprecated since version 2.0: Use *T* instead.

classmethod zero(*shape=(1, 1)*)

Return a constant zero expression of given shape.

property H

Conjugate (or Hermitian) transpose.

property T

Matrix transpose.

property T0

Expression with the first 2×2 subsystem transposed.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_transpose* otherwise.

property T1

Expression with the second 2×2 subsystem transposed.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_transpose* otherwise.

property T2

Expression with the third 2×2 subsystem transposed.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_transpose* otherwise.

property T3

Expression with the fourth 2×2 subsystem transposed.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_transpose* otherwise.

property Tl

Expression with the last 2×2 subsystem transposed.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_transpose* otherwise.

property Tx

Auto-detect few subsystems of same shape and transpose the last.

Deprecated since version 2.0: Use *partial_transpose* instead.

property bilin

Pure bilinear part of the expression.

property colsum

Sum over the columns of the expression as a column vector.

property complex

Whether the expression can be complex-valued.

property conj

Complex conjugate.

property cst

Constant part of the expression.

property desvec

The reverse operation of *svec*.

Raises

- **TypeError** – If the expression is not a vector or has a dimension outside the permissible set

$$\left\{ \frac{n(n+1)}{2} \mid n \in \mathbb{Z}_{\geq 1} \right\} = \left\{ n \in \mathbb{Z}_{\geq 1} \mid \frac{1}{2} \left(\sqrt{8n+1} - 1 \right) \in \mathbb{Z}_{\geq 1} \right\}.$$

- **ValueError** – In the case of a complex vector, If the vector is not in the image of *svec*.

property diag

Diagonal matrix with the expression on the main diagonal.

Vectorization is performed in column-major order.

property hermitian

Whether the expression is a hermitian (or symmetric) matrix.

Uses *RELATIVE_HERMITIANNESS_TOLERANCE*.

If PICOS rejects your near-hermitian (near-symmetric) expression as not hermitian (not symmetric), you can use *hermitianized* to correct larger numeric errors or the effects of noisy data.

property hermitianized

The expression projected onto the subspace of hermitian matrices.

If the expression is not complex, then this is a projection onto the subspace of symmetric matrices.

This is the same as *opreal*.

property imag

Imaginary part of the expression.

property is0

Whether this is a constant scalar, vector or matrix of all zeros.

property is1

Whether this is a constant scalar or vector of all ones.

property isI

Whether this is a constant identity matrix.

property isreal

Whether the expression is always real-valued.

property lin

Linear part of the expression.

property maindiag

The main diagonal of the expression as a column vector.

property noncst

Nonconstant part of the expression.

property opimag

The imaginary part of a Hermitian operator.

For a square (complex) affine expression A, this is $\frac{1}{2i}(A - A^H)$.

Example

```
>>> from picos import ComplexVariable
>>> ComplexVariable("X", (4, 4)).opimag.hermitian
True
```

property opreal

The real part of a Hermitian operator.

For a square (complex) affine expression A, this is $\frac{1}{2}(A + A^H)$.

Example

```
>>> from picos import ComplexVariable
>>> ComplexVariable("X", (4, 4)).opreal.hermitian
True
```

property real

Real part of the expression.

property rowsum

Sum over the rows of the expression as a row vector.

property sum

Sum over all scalar elements of the expression.

property svec

An isometric vectorization of a symmetric or Hermitian expression.

In the real symmetric case

- the vectorization format is precisely the one define in [svec],
- the vectorization is isometric and isomorphic, and
- this is the same vectorization as used internally by the *SymmetricVariable* class.

In the complex hermitian case

- the same format is used, now resulting in a complex vector,
- the vectorization is isometric but **not** isomorphic as there are guaranteed zeros in the imaginary part of the vector, and
- this is **not** the same vectorization as the isomorphic, real-valued one used by *HermitianVariable*.

The reverse operation is denoted by *desvec* in either case.

Raises

- `TypeError` – If the expression is not square.
- `ValueError` – If the expression is not hermitian.

property tr

Trace of a square expression.

property tr0

Expression with the first 2×2 subsystem traced out.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_trace* otherwise.

property tr1

Expression with the second 2×2 subsystem traced out.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_trace* otherwise.

property tr2

Expression with the third 2×2 subsystem traced out.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_trace* otherwise.

property tr3

Expression with the fourth 2×2 subsystem traced out.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_trace* otherwise.

property trilvec

Column-major vectorization of the lower triangular part.

Returns

A column vector of all elements A_{ij} that satisfy $i \geq j$.

Note: If you want a row-major vectorization instead, write `A.T.triuvec` instead of `A.trilvec`.

Example

```
>>> from picos import Constant
>>> A = Constant("A", [[1, 2], [3, 4], [5, 6]])
>>> print(A)
[ 1.00e+00  2.00e+00]
[ 3.00e+00  4.00e+00]
[ 5.00e+00  6.00e+00]
>>> print(A.trilvec)
[ 1.00e+00]
[ 3.00e+00]
[ 5.00e+00]
[ 4.00e+00]
[ 6.00e+00]
```

property triuvec

Column-major vectorization of the upper triangular part.

Returns
A column vector of all elements A_{ij} that satisfy $i \leq j$.

Note: If you want a row-major vectorization instead, write `A.T.trilvec` instead of `A.triuvec`.

Example

```
>>> from picos import Constant
>>> A = Constant("A", [[1, 2, 3], [4, 5, 6]])
>>> print(A)
[ 1.00e+00  2.00e+00  3.00e+00]
[ 4.00e+00  5.00e+00  6.00e+00]
>>> print(A.triuvec)
[ 1.00e+00]
[ 2.00e+00]
[ 5.00e+00]
[ 3.00e+00]
[ 6.00e+00]
```

property trl

Expression with the last 2×2 subsystem traced out.

Only available for a $2^k \times 2^k$ matrix with all subsystems of shape 2×2. Use *partial_trace* otherwise.

property vec

Column-major vectorization of the expression as a column vector.

Note: Given an expression `A`, `A.vec` and `A[:]` produce the same result (up to its string description) but `A.vec` is faster and its result is cached.

Example

```
>>> from picos import Constant
>>> A = Constant("A", [[1, 2], [3, 4]])
>>> A.vec.equals(A[:])
True
```

(continues on next page)

(continued from previous page)

```
>>> A[:] is A[:]
False
>>> A.vec is A.vec
True
```

picos.expressions.exp_detrootn

Implements *DetRootN*.

Classes

class picos.expressions.exp_detrootn.**DetRootN**(*x*)

Bases: *Expression*

The n-th root of the determinant of an $n \times n$ matrix.

Definition

For an $n \times n$ positive semidefinite hermitian matrix X, this is

$$\sqrt[n]{\det X}.$$

Warning: When you pose a lower bound on the n-th root of a determinant of the matrix X, then PICOS enforces positive semidefiniteness $X \succeq 0$ through an auxiliary constraint during solution search.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*x*)

Construct a *DetRootN*.

Parameters

x (ComplexAffineExpression) – The matrix concerned. Must be hermitian by definition.

__mul__(*other*)

Denote multiplication with another expression on the right.

__rmul__(*other*)

Denote multiplication with another expression on the left.

property n

Diagonal length of *x*.

property x

The matrix concerned.

picos.expressions.exp_entropy

Implements *Entropy* and *NegativeEntropy*.

Classes

class picos.expressions.exp_entropy.**Entropy**(*x*, *y=None*)

Bases: *Expression*

Entropy or negative relative entropy of an affine expression.

Negative relative entropy is also known as the perspective of the logarithm.

Definition

Let x be an n-dimensional real affine expression.

1. If no additional expression y is given, this is the entropy

$$-\sum_{i=1}^{n} \operatorname{vec}(x)_i \log(\operatorname{vec}(x)_i).$$

2. If an additional affine expression y of same shape as x is given, this is the negative relative entropy (or logarithmic perspective)

$$\begin{aligned}-\sum_{i=1}^{n} \operatorname{vec}(x)_i \log\left(\frac{\operatorname{vec}(x)_i}{\operatorname{vec}(y)_i}\right) &= -\sum_{i=1}^{n} \operatorname{vec}(x)_i \left[\log(\operatorname{vec}(x)_i) - \log(\operatorname{vec}(y)_i)\right] \\ &= \sum_{i=1}^{n} \operatorname{vec}(x)_i \left[\log(\operatorname{vec}(y)_i) - \log(\operatorname{vec}(x)_i\right] \\ &= \sum_{i=1}^{n} \operatorname{vec}(x)_i \log\left(\frac{\operatorname{vec}(y)_i}{\operatorname{vec}(x)_i}\right).\end{aligned}$$

Warning: When you pose a lower bound on this expression, then PICOS enforces $x \geq 0$ through an auxiliary constraint during solution search. When an additional expression y is given, PICOS enforces $y \geq 0$ as well.

__add__(*other*)

Denote addition with another expression on the right-hand side.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*x*, *y=None*)

Construct an *Entropy*.

Parameters

- **x** (AffineExpression) – The affine expression x.
- **y** (AffineExpression) – An additional affine expression y. If necessary, PICOS will attempt to reshape or broadcast it to the shape of x.

__neg__()

Denote the negation of the expression.

__radd__(*other*)

Denote addition with another expression on the left-hand side.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

`property n`

Length of `x`.

`property x`

The expression x.

`property y`

The additional expression y, or `None`.

`class picos.expressions.exp_entropy.`**`NegativeEntropy`**(*x*, *y=None*)

Bases: *`Expression`*

Negative or relative entropy of an affine expression.

Relative entropy is also known as the Kullback-Leibler divergence.

Definition

Let x be an n-dimensional real affine expression.

1. If no additional expression y is given, this is the negative entropy

$$\sum_{i=1}^{n} \operatorname{vec}(x)_i \log(\operatorname{vec}(x)_i).$$

2. If an additional affine expression y of same shape as x is given, this is the relative entropy (or Kullback-Leibler divergence)

$$\sum_{i=1}^{n} \operatorname{vec}(x)_i \log\left(\frac{\operatorname{vec}(x)_i}{\operatorname{vec}(y)_i}\right) = \sum_{i=1}^{n} \operatorname{vec}(x)_i \left[\log(\operatorname{vec}(x)_i) - \log(\operatorname{vec}(y)_i)\right].$$

Warning: When you pose an upper bound on this expression, then PICOS enforces $x \geq 0$ through an auxiliary constraint during solution search. When an additional expression y is given, PICOS enforces $y \geq 0$ as well.

`__add__`(*other*)

Denote addition with another expression on the right-hand side.

`__init__`(*x*, *y=None*)

Construct a *`NegativeEntropy`*.

Parameters

- **x** (`AffineExpression`) – The affine expression x.
- **y** (`AffineExpression`) – An additional affine expression y. If necessary, PICOS will attempt to reshape or broadcast it to the shape of x.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__neg__`()

Denote the negation of the expression.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

`property n`

Length of `x`.

`property x`

The expression x.

`property y`

The additional expression y, or `None`.

picos.expressions.exp_extremum

Implements *`MaximumConvex`* and *`MinimumConcave`*.

Classes

`class picos.expressions.exp_extremum.`**`Extremum`**(*expressions*)

Bases: *`ExtremumBase`*, *`Expression`*

Base class for *`MaximumConvex`* and *`MinimumConcave`*.

Note: This can represent the maximum (minimum) over convex (concave) uncertain expressions as long as the uncertainty is not of stochastic nature. In this case, the extremum implicitly goes over the perturbation parameters as well.

`__ge__`(*other*)

Return self>=value.

`__init__`(*expressions*)

Construct a *`MaximumConvex`* or *`MinimumConcave`*.

Parameters

`expressions` – A collection of all convex or all concave expressions.

`__le__`(*other*)

Return self<=value.

`property expressions`

The expressions under the extremum.

`class picos.expressions.exp_extremum.`**`ExtremumBase`**

Bases: `ABC`

Base class for *`Extremum`* and similar classes.

In particular, this is also used by the uncertain *`RandomExtremumAffine`*.

Must be inherited with priority with respect to *`Expression`*.

`__mul__`(*other*)

`__neg__`()

`__rmul__`(*other*)

`property argnum`

Number of expressions under the extremum.

abstract property expressions

The expressions under the extremum.

class picos.expressions.exp_extremum.MaximumBase

Bases: `object`

Base implementation of *ExtremumBase* for maximums.

class picos.expressions.exp_extremum.MaximumConvex(*expressions*)

Bases: *MaximumBase*, *Extremum*

The maximum over a set of convex scalar expressions.

Example

```
>>> import picos
>>> x = picos.RealVariable("x", 4)
>>> a = abs(x)
>>> b = picos.sum(x)
>>> c = picos.max([a, b]); c
<Maximum of Convex Functions: max(‖x‖, ∑(x))>
>>> 2*c
<Scaled Maximum of Convex Functions: 2·max(‖x‖, ∑(x))>
>>> c <= 5
<Maximum of Convex Functions Constraint: max(‖x‖, ∑(x)) ≤ 5>
```

class picos.expressions.exp_extremum.MinimumBase

Bases: `object`

Base implementation of *ExtremumBase* for minimums.

class picos.expressions.exp_extremum.MinimumConcave(*expressions*)

Bases: *MinimumBase*, *Extremum*

The minimum over a set of concave scalar expressions.

Example

```
>>> import picos
>>> x = picos.RealVariable("x", 4)
>>> a = picos.sum(x)
>>> b = 2*a
>>> c = picos.min([a, b]); c
<Minimum of Concave Functions: min(∑(x), 2·∑(x))>
>>> -1*c
<Maximum of Convex Functions: max(-∑(x), -2·∑(x))>
>>> C = 5 <= c; C
<Minimum of Concave Functions Constraint: min(∑(x), 2·∑(x)) ≥ 5>
>>> x.value = 1
>>> C.slack
-1.0
```

picos.expressions.exp_geomean

Implements *GeometricMean*.

Classes

class picos.expressions.exp_geomean.**GeometricMean**(*x*)

Bases: *Expression*

Geometric mean of an affine expression.

Definition

For an n-dimensional affine expression x with $x \geq 0$, the geometric mean is given as

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}.$$

Warning: When you pose a lower bound on a geometric mean, then PICOS enforces $x \geq 0$ through an auxiliary constraint during solution search.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*x*)

Construct a *GeometricMean*.

Parameters

x (AffineExpression) – The affine expression to form the geometric mean of.

__mul__(*other*)

Denote multiplication with another expression on the right.

__rmul__(*other*)

Denote multiplication with another expression on the left.

property x

The expression under the mean.

picos.expressions.exp_logarithm

Implements *Logarithm*.

Classes

class picos.expressions.exp_logarithm.**Logarithm**(*x*)

Bases: *Expression*

Logarithm of a scalar affine expression.

Definition

For a real scalar affine expression x, this is $\log(x)$.

Warning: When you pose a lower bound on a logarithm $\log(x)$, then PICOS enforces $x \geq 0$ through an auxiliary constraint during solution search.

`__add__`(*other*)

Denote addition with another expression on the right-hand side.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*x*)

Construct a *`Logarithm`*.

Parameters

x (`AffineExpression`) – The scalar affine expression x.

`__mul__`(*other*)

Denote multiplication with another expression on the right.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__rmul__`(*other*)

Denote multiplication with another expression on the left.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

`property` **`exp`**

The exponential of the logarithm, equal to x.

`property` **`x`**

The expression x.

picos.expressions.exp_logsumexp

Implements *`LogSumExp`*.

Classes

`class picos.expressions.exp_logsumexp.`**`LogSumExp`**(*x*)

Bases: *`Expression`*

Logarithm of the sum of elementwise exponentials of an expression.

Definition

For an n-dimensional real affine expression x, this is the logarithm of the sum of elementwise exponentials

$$\log \sum_{i=1}^{n} \exp(\operatorname{vec}(x)_i).$$

`__add__`(*other*)

Denote addition with another expression on the right-hand side.

`__init__`(*x*)

Construct a *`LogSumExp`*.

Parameters

x (`AffineExpression`) – The affine expression x.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

`property exp`

The elementwise sum of exponentials of x.

`property n`

Length of `x`.

`property x`

The expression x.

picos.expressions.exp_mtxgeomean

Implements `MatrixGeometricMean`.

Classes

`class picos.expressions.exp_mtxgeomean.`**`MatrixGeometricMean`**(*X*, *Y*, *power=0.5*)

Bases: `Expression`

Matrix geometric mean of an affine expression.

Definition

For $n \times n$-dimensional symmetric or Hermitian matrices X and Y, this is defined as

$$X^{1/2}(X^{-1/2}Y^{-1}X^{-1/2})^pX^{1/2}.$$

for a given scalar $p \in [-1, 2]$, where $p = 1/2$ by default.

Warning: When you pose an upper or lower bound on this expression, then PICOS enforces $X \succeq 0$ and $Y \succeq 0$ through an auxiliary constraint during solution search.

`__init__`(*X*, *Y*, *power=0.5*)

Construct an `MatrixGeometricMean`.

Parameters

- **X** (`AffineExpression`) – The affine expression X.
- **Y** (`AffineExpression`) – The affine expression Y. This should have the same dimensions as X.

`property X`

The expression X.

`property Y`

The additional expression Y.

`property iscomplex`

Whether `X` and `Y` are complex expressions or not.

`property n`

Lengths of `X` and `Y`.

property power

The power p.

property tr

Trace of the matrix geometric mean.

class picos.expressions.exp_mtxgeomean.**TrMatrixGeometricMean**(*X*, *Y*, *power=0.5*)

Bases: `MatrixGeometricMean`

Trace matrix geometric mean of an affine expression.

Definition

For $n \times n$-dimensional symmetric or Hermitian matrices X and Y, this is defined as

$$\operatorname{Tr}(X^{1/2}(X^{-1/2}Y^{-1}X^{-1/2})^{p}X^{1/2}).$$

for a given scalar $p \in [-1, 2]$, where $p = 1/2$ by default.

Warning: When you pose an upper or lower bound on this expression, then PICOS enforces $X \succeq 0$ and $Y \succeq 0$ through an auxiliary constraint during solution search.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*X*, *Y*, *power=0.5*)

Construct an `MatrixGeometricMean`.

Parameters

- **X** (`AffineExpression`) – The affine expression X.
- **Y** (`AffineExpression`) – The affine expression Y. This should have the same dimensions as X.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

picos.expressions.exp_norm

Implements `Norm`.

Classes

class picos.expressions.exp_norm.**Norm**(*x*, *p=2*, *q=None*, *denominator_limit=1000*)

Bases: `Expression`

Entrywise p-norm or $L_{p,q}$-norm of an expression.

This class can represent the absolute value, the modulus, a vector p-norm and an entrywise matrix $L_{p,q}$-norm of a (complex) affine expression. In addition to these convex norms, it can represent the concave *generalized vector p-norm* with $0 < p \leq 1$.

Not all of these norms are available on the complex field; see the definitions below to learn more.

Definition

If q is not given (`None`), then it is set equal to p.

1. If the normed expression is a real scalar x, then this is the absolute value

   $$|x| = \begin{cases} x, & \text{if } x \geq 0, \\ -x, & \text{otherwise.} \end{cases}$$

   The parameters p and q are ignored in this case.

2. If the normed expression is a complex scalar z, then this is the modulus

   $$|z| = \sqrt{\operatorname{Re}(z)^2 + \operatorname{Im}(z)^2}.$$

   The parameters p and q are ignored in this case.

3. If the normed expression is a real vector x and $p = q$, then this is the (generalized) vector p-norm

   $$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$$

   for $p \in \mathbb{Q} \cup \{\infty\}$ with $p > 0$ and $|x_i|$ the absolute value of the i-th entry of x.

   Note that for $p < 1$ the expression is not convex and thus not a proper norm. However, it is concave over the nonnegative orthant and posing a lower bound on such a generalized norm yields a convex constraint for $x \geq 0$.

   Warning: When you pose a lower bound on a concave generalized norm ($p < 1$), then PICOS enforces $x \geq 0$ through an auxiliary constraint during solution search.

   Special cases:

   - For $p = 1$, this is the *Manhattan* or *Taxicab* norm $\|x\|_{\text{sum}}$.
   - For $p = 2$, this is the Euclidean norm $\|x\| = \|x\|_2$.
   - For $p = \infty$, this is the *Maximum*, *Chebyshev*, or *Infinity* norm $\|x\|_{\text{max}}$.

4. If the normed expression is a real vector x and $p \neq q$, then it is treated as a matrix with a single row or a single column, depending on the shape associated with x. See case (5).

5. If the normed expression is a complex vector z and $p = q$, then the definition is the same as in case (3) but with $x = z$, $|x_i|$ the modulus of the i-th entry of x, and $p \geq 1$.

6. If the normed expression is a real $m \times n$ matrix X, then this is the $L_{p,q}$-norm

   $$\|X\|_{p,q} = \left(\sum_{j=1}^{n} \left(\sum_{i=1}^{m} |X_{ij}|^p\right)^{\frac{q}{p}}\right)^{\frac{1}{q}}$$

   for $p, q \in \mathbb{Q} \cup \{\infty\}$ with $p, q \geq 1$.

   If $p = q$, then this is equal to the (generalized) vector p-norm of the the vectorized matrix, that is $\|X\|_{p,p} = \|\operatorname{vec}(X)\|_p$. In this case, the requirement $p \geq 1$ is relaxed to $p > 0$ and X may be a complex matrix. See case (3).

   Special cases:

   - For $p = q = 2$, this is the Frobenius norm $\|X\| = \|X\|_F$.
   - For $p = 1$, $q = \infty$, this is the maximum absolute column sum

     $$\|X\|_{1,\infty} = \max_{j=1 \ldots n} \sum_{i=1}^{m} |X_{ij}|.$$

     This equals the operator norm induced by the vector 1-norm. You can obtain the maximum absolute row sum (the operator norm induced by the vector ∞-norm) by first transposing X.

7. Complex matrix norms are not supported.

Note: You can write ∞ in Python as `float("inf")`.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*x*, *p=2*, *q=None*, *denominator_limit=1000*)

Construct a `Norm`.

Parameters

- **x** (`ComplexAffineExpression`) – The affine expression to take the norm of.
- **p** (*`float`*) – The value for p, which is cast to a limited precision fraction.
- **q** (*`float`*) – The value for q, which is cast to a limited precision fraction. The default of `None` means *equal to* p.
- **denominator_limit** (*`int`*) – The largest allowed denominator when casting p and q to a fraction. Higher values can yield a greater precision at reduced performance.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__mul__`(*other*)

Denote multiplication with another expression on the right.

`__pow__`(*other*)

Denote exponentiation with another, scalar expression.

`__rmul__`(*other*)

Denote multiplication with another expression on the left.

property p

The parameter p.

This is a limited precision version of the parameter used when the norm was constructed.

property pden

The limited precision fraction denominator of p.

property pnum

The limited precision fraction numerator of p.

property q

The parameter q.

This is a limited precision version of the parameter used when the norm was constructed.

property qden

The limited precision fraction denominator of q.

property qnum

The limited precision fraction numerator of q.

property x

Real expression whose norm equals that of the original expression.

picos.expressions.exp_nucnorm

Implements *NuclearNorm*.

Classes

class picos.expressions.exp_nucnorm.**NuclearNorm**(*x*)

Bases: *Expression*

The nuclear norm of a matrix.

This class can represent the nuclear norm of a matrix-affine expression (real- or complex valued). The nuclear norm is convex, so we can form expressions of the form `NuclearNorm(X) <= t` which are typically reformulated as LMIs that can be handled by SDP solvers.

Definition

If the normed expression is a matrix X, then its nuclear norm is

$$\|X\|_* = \operatorname{trace}\left(X^* X\right)^{1/2} = \sum_{i=1}^{\min(n,m)} \sigma_i(X)$$

where the $\sigma_i(X)$ denote the singular values of a X, and X^* denotes the adjoint matrix of X (i.e., the transposed matrix X^T if X is real-valued).

Special cases:

- If X is scalar, then $\|X\|_*$ reduces to the the absolute value (or modulus) $|X|$.
- If X is scalar, then $\|X\|_*$ coincides with the Euclidean norm of X.

__init__(*x*)

Construct a *NuclearNorm*.

Parameters

x (`ComplexAffineExpression`) – The affine expression to take the norm of.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

__mul__(*other*)

Denote multiplication with another expression on the right.

__rmul__(*other*)

Denote multiplication with another expression on the left.

property x

Real expression whose norm equals that of the original expression.

picos.expressions.exp_oprelentr

Implements *OperatorRelativeEntropy*.

Classes

class picos.expressions.exp_oprelentr.**OperatorRelativeEntropy**(X, Y)

Bases: *Expression*

Operator relative entropy of an affine expression.

Definition

For $n \times n$-dimensional symmetric or Hermitian matrices X and Y, this is defined as

$$X^{1/2} \log(X^{1/2} Y^{-1} X^{1/2}) X^{1/2}.$$

Warning: When you pose an upper bound on this expression, then PICOS enforces $X \succeq 0$ and $Y \succeq 0$ through an auxiliary constraint during solution search.

__init__(X, Y)

Construct an *OperatorRelativeEntropy*.

Parameters

- **X** (AffineExpression) – The affine expression X.
- **Y** (AffineExpression) – The affine expression Y. This should have the same dimensions as X.

property X

The expression X.

property Y

The additional expression Y.

property iscomplex

Whether *X* and *Y* are complex expressions or not.

property n

Lengths of *X* and *Y*.

property tr

Trace of the operator relative entropy.

class picos.expressions.exp_oprelentr.**TrOperatorRelativeEntropy**(X, Y)

Bases: *OperatorRelativeEntropy*

Trace operator relative entropy of an affine expression.

Definition

For $n \times n$-dimensional symmetric or Hermitian matrices X and Y, this is defined as

$$\mathrm{Tr}(X^{1/2} \log(X^{1/2} Y^{-1} X^{1/2}) X^{1/2}).$$

Warning: When you pose an upper bound on this expression, then PICOS enforces $X \succeq 0$ and $Y \succeq 0$ through an auxiliary constraint during solution search.

__init__(X, Y)

Construct an *OperatorRelativeEntropy*.

Parameters

- **X** (AffineExpression) – The affine expression X.
- **Y** (AffineExpression) – The affine expression Y. This should have the same dimensions as X.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

picos.expressions.exp_powtrace

Implements *PowerTrace*.

Classes

`class picos.expressions.exp_powtrace.`**`PowerTrace`**(*x*, *p*, *m=None*, *denominator_limit=1000*)

Bases: *Expression*

The trace of the p-th power of a hermitian matrix.

Definition

Let $p \in \mathbb{Q}$.

1. If the base expressions is a real scalar x and no additional constant m is given, then this is the power x^p.
2. If the base expressions is a real scalar x, $p \in [0, 1]$, and a positive scalar constant m is given, then this is the scaled power mx^p.
3. If the base expression is a hermitian matrix X and no additional constant M is given, then this is the trace of power $\operatorname{tr}(X^p)$.
4. If the base expression is a hermitian matrix X, $p \in [0, 1]$, and a hermitian positive semidefinite constant matrix M of same shape as X is given, then this is the trace of a scaled power $\operatorname{tr}(MX^p)$.

No other case is supported. In particular, if $p \notin [0, 1]$, then m/M must be undefined (`None`).

Warning:

1. For a constraint of the form $x^p \leq t$ with $p < 1$ and $p \neq 0$, PICOS enforces $x \geq 0$ during solution search.
2. For a constraint of the form $\operatorname{tr}(X^p) \leq t$ or $\operatorname{tr}(MX^p) \leq t$ with $p < 1$ and $p \neq 0$, PICOS enforces $X \succeq 0$ during solution search.
3. For a constraint of the form $\operatorname{tr}(X^p) \leq t$ or $\operatorname{tr}(MX^p) \leq t$ with $p > 1$, PICOS enforces $t \geq 0$ during solution search.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*x*, *p*, *m=None*, *denominator_limit=1000*)

Construct a *PowerTrace*.

Parameters

- **x** (AffineExpression) – The scalar or symmetric matrix to form a power of.
- **p** (*float*) – The value for p, which is cast to a limited precision fraction.
- **m** (*AffineExpression* or anything recognized by *load_data*) – An additional positive semidefinite constant to multiply the power with.

- **denominator_limit** (*int*) – The largest allowed denominator when casting p to a fraction. Higher values can yield a greater precision at reduced performance.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

__mul__(*other*)

Denote multiplication with another expression on the right.

__rmul__(*other*)

Denote multiplication with another expression on the left.

property den

The limited precision fraction denominator of p.

property m

An additional factor to multiply the power with.

property n

Diagonal length of `x`.

property num

The limited precision fraction numerator of p.

property p

The parameter p.

This is a limited precision version of the parameter used when the expression was constructed.

property x

The matrix concerned.

picos.expressions.exp_quadratic

Implements `QuadraticExpression`.

Classes

class picos.expressions.exp_quadratic.**QuadraticExpression**(*string*, *quadraticPart={}*, *affinePart=AffineExpression.zero()*, *scalarFactors=None*, *copyDecomposition=None*)

Bases: `Expression`

A scalar quadratic expression of the form $x^T Qx + a^T x + b$.

__add__(*other*)

Denote addition from the right hand side.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*string*, *quadraticPart={}*, *affinePart=AffineExpression.zero()*, *scalarFactors=None*, *copyDecomposition=None*)

Initialize a scalar quadratic expression.

This constructor is meant for internal use. As a user, you will most likely want to build expressions starting from `variables` or a `Constant`.

Parameters

- **string** (*str*) – A symbolic string description.

- **quadraticPart** – A dict mapping PICOS variable pairs to CVXOPT matrices. Each entry $(x, y) \mapsto A_{xy}$ represents a quadratic form $\operatorname{vec}_x(x)^T A_{xy} \operatorname{vec}_y(y)$ where vec_x and vec_y refer to the isometric real *vectorizations* that are used by PICOS internally to store the variables x and y, respectively. The quadratic part Q of the expression is then given as $Q = \sum_{x,y} A_{xy}$.
- **affinePart** (AffineExpression) – The affine part $a^T x + b$ of the expression.
- **scalarFactors** – A pair (u, v) with both u and v scalar real affine expressions representing a known $x^T Q x + a^T x + b = uv$ factorization of the expression.
- **copyDecomposition** (QuadraticExpression) – Another quadratic expression with equal quadratic part whose quadratic part decomposition shall be copied.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

__mul__(*other*)

Denote scaling from the right hand side.

__neg__()

Denote the negation of the expression.

__radd__(*other*)

Denote addition from the left hand side.

__rmul__(*other*)

Denote scaling from the left hand side.

__rsub__(*other*)

Denote substraction with self on the right hand side.

__sub__(*other*)

Denote substraction from the right hand side.

__truediv__(*other*)

Denote division by a constant scalar.

property A

An affine-augmented quadratic coefficient matrix, condensed.

For a quadratic expression $x^T Q x + a^T x + b$, this is $A = \begin{bmatrix} Q & \frac{a}{2} \\ \frac{a^T}{2} & b \end{bmatrix}$ but with zero rows and columns removed.

The vector *y* is a condensed version of x that refers to this matrix, so that `q.y.T*q.A*q.y` equals the expression `q`.

Raises

ValueError – When the expression is zero.

property L

The L of an LL^T Cholesky decomposition of *Q*.

Returns

A CVXOPT lower triangular sparse matrix.

Raises

- **ValueError** – When the quadratic part is zero.
- **ArithmeticError** – When the expression is not convex, that is when the matrix *Q* is not numerically positive semidefinite.

property M

The M of an MM^T Cholesky decomposition of `A`.

Returns

A CVXOPT lower triangular sparse matrix.

Raises

- `ValueError` – When the expression is zero.
- `ArithmeticError` – When the expression can not be written as a squared norm, that is when the extended matrix `A` is not numerically positive semidefinite.

property Q

The coefficient matrix Q of the expression, condensed.

This equals the Q of the quadratic expression x^TQx+a^Tx+b but with zero rows and columns removed.

The vector `x` is a condensed version of x that refers to this matrix, so that `q.x.T*q.Q*q.x` equals the quadratic part x^TQx of the expression `q`.

Raises

`ValueError` – When the quadratic part is zero.

property aff

Affine part $a^Tx + b$ of the quadratic expression.

property cst

Constant part b of the quadratic expression.

property fullroot

Affine expression whose squared norm equals the expression.

For a convex quadratic expression `q`, this is equal to the vector `q.M.T*q.y` with zero rows removed.

Construction

For a quadratic expression x^TQx+a^Tx+b with $A = \begin{bmatrix} Q & \frac{a}{2} \\ \frac{a^T}{2} & b \end{bmatrix}$ positive semidefinite, let $A = MM^T$ be a Cholesky decomposition. Let further $y = \begin{bmatrix} x^T & 1 \end{bmatrix}^T$. Then,

$$\begin{aligned} x^TQx + a^Tx + b &= y^TAy \\ &= y^TMM^Ty \\ &= (M^Ty)^TM^Ty \\ &= \langle M^Ty, M^Ty \rangle \\ &= \|M^Ty\|^2. \end{aligned}$$

Note that removing zero rows from M^Ty does not affect the norm.

Raises

- `ValueError` – When the expression is zero.
- `ArithmeticError` – When the expression is not convex, that is when the quadratic part is not numerically positive semidefinite.

property is0

Whether the quadratic expression is zero.

property is_squared_norm

Whether the expression can be written as a squared norm.

If this is `True`, then the there is a coefficient vector c such that

$$\|c^T \begin{bmatrix} x \\ 1 \end{bmatrix}\|^2 = x^TQx + a^Tx + b.$$

If the expression is also nonzero, then *fullroot* is $c^T \begin{bmatrix} x^T & 1 \end{bmatrix}^T$ with zero entries removed.

property lin

Linear part $a^T x$ of the quadratic expression.

property num_quad_terms

The number of terms in the simplified quadratic form.

property quad

Quadratic part $x^T Q x$ of the quadratic expression.

property quadratic_forms

The quadratic forms as a map from variable pairs to sparse matrices.

Warning: Do not modify the returned matrices.

Deprecated since version 2.2: This property will be removed in a future release.

property quadroot

Affine expression whose squared norm equals $x^T Q x$.

For a convex quadratic expression `q`, this is equal to the vector `q.L.T*q.x` with zero rows removed.

Construction

Let $x^T Q x$ be the quadratic part of the expression with Q positive semidefinite and $Q = LL^T$ a Cholesky decomposition. Then,

$$\begin{aligned} x^T Q x &= x^T L L^T x \\ &= (L^T x)^T L^T x \\ &= \langle L^T x, L^T x \rangle \\ &= \|L^T x\|^2. \end{aligned}$$

Note that removing zero rows from $L^T x$ does not affect the norm.

Raises

- `ValueError` – When the quadratic part is zero.
- `ArithmeticError` – When the expression is not convex, that is when the quadratic part is not numerically positive semidefinite.

property rank

The length of the vector *quadroot*.

Up to numerical considerations, this is the rank of the (convex) quadratic coefficient matrix Q of the expression.

Raises

`ArithmeticError` – When the expression is not convex, that is when the quadratic part is not numerically positive semidefinite.

property scalar_factors

Decomposition into scalar real affine expressions.

If the expression is known to be equal to ab for scalar real affine expressions a and b, this is the pair `(a, b)`. Otherwise, this is `None`.

Note that if $a = b$, then also `a is b` in the returned tuple.

property x

The stacked variable vector x of the expression, condensed.

This equals the x of the quadratic expression $x^T Q x + a^T x + b$ but entries corresponding to zero rows and columns in `Q` are removed, so that `q.x.T*q.Q*q.x` equals the $x^T Q x$ part of the expression `q`.

Raises

`ValueError` – When the quadratic part is zero.

property y

See `A`.

Raises

`ValueError` – When the expression is zero.

Objects

`picos.expressions.exp_quadratic.`**`PSD_PERTURBATIONS`**

Maximum number of singular quadratic form perturbations.

PICOS uses NumPy either or CHOLMOD to compute a Cholesky decomposition of a positive semidefinite quadratic form Q. Both libraries require Q to be nonsingular, which is not a requirement on PICOS' end. If either library rejects Q, then PICOS provides a sequence of *`PSD_PERTURBATIONS`* perturbed quadratic forms $Q + \epsilon I$ for increasing ϵ until the perturbed matrix is found positive definite or until the largest ϵ was tested unsuccessfully.

If this is zero, PICOS will only decompose quadratic forms that are nonsingular. If this is one, then only the largest epsilon is tested.

Default value

```
3
```

picos.expressions.exp_quantcondentr

Implements *`QuantumConditionalEntropy`*.

Classes

class `picos.expressions.exp_quantcondentr.`**`QuantumConditionalEntropy`**(*X*, *subsystems*, *dimensions=2*)

Bases: *`Expression`*

Quantum conditional entropy of an affine expression.

Definition

Let X be an $N \times N$-dimensional symmetric or hermitian matrix. Then this is defined as

$$-S(X) + S(\operatorname{Tr}_i(X)),$$

where $S(X) = -\operatorname{Tr}(X \log(X))$ is the quantum entropy, and Tr_i denotes the partial trace with respect to the i-th subsystem.

Warning: When you pose a lower bound on this expression, then PICOS enforces $X \succeq 0$ through an auxiliary constraint during solution search.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*X*, *subsystems*, *dimensions=2*)

Construct an `QuantumConditionalEntropy`.

Parameters

- **X** (`AffineExpression`) – The affine expression X.
- **subsystems** (*int or tuple or list*) – A collection of or a single subystem number, indexed from zero, corresponding to subsystems that shall be traced over. The value -1 refers to the last subsystem.
- **dimensions** (*int or tuple or list*) – Either an integer d so that the subsystems are assumed to be all of shape $d \times d$, or a sequence of subsystem shapes where an integer d within the sequence is read as $d \times d$. In any case, the elementwise product over all subsystem shapes must equal the expression's shape.

property X

The expression X.

property dimensions

The dimensions of the subsystems of X.

property iscomplex

Whether `X` is a complex expression or not.

property n

Length of `X`.

property subsystems

The subsystems being traced out of X.

picos.expressions.exp_quantentr

Implements `QuantumEntropy`, `NegativeQuantumEntropy`.

Classes

`class picos.expressions.exp_quantentr.`**`NegativeQuantumEntropy`**(*X*, *Y=None*)

Bases: `Expression`

Negative or quantum relative entropy of an affine expression.

Definition

Let X be an $n \times n$-dimensional symmetric or hermitian matrix.

1. If no additional expression Y is given, this is the negative quantum entropy

$$\mathrm{Tr}(X \log(X)).$$

2. If an additional affine expression Y of same shape as X is given, this is the quantum relative entropy

$$\mathrm{Tr}(X \log(X) - X \log(Y)).$$

3. If an additional scalar valued real affine expression Y is given, this is the homogenized negative quantum entropy

$$\mathrm{Tr}(X \log(X/y)).$$

> **Warning:** When you pose an upper bound on this expression, then PICOS enforces $X \succeq 0$ through an auxiliary constraint during solution search. When an additional expression Y is given, PICOS enforces $Y \succeq 0$ as well.

`__init__`(*X*, *Y=None*)

Construct a `NegativeQuantumEntropy`.

Parameters

- **X** (`AffineExpression`) – The affine expression X.
- **Y** (`AffineExpression`) – An additional affine expression Y. If necessary, PICOS will attempt to reshape or broadcast it to the shape of X.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__neg__`()

Denote the negation of the expression.

property X

The expression X.

property Y

The additional expression Y, or `None`.

property iscomplex

Whether `X` and `Y` are complex expressions or not.

property n

Length of `X`.

class `picos.expressions.exp_quantentr.`**`QuantumEntropy`**(*X*, *Y=None*)

Bases: `Expression`

Quantum or negative quantum relative entropy of an affine expression.

Definition

Let X be an $n \times n$-dimensional symmetric or hermitian matrix.

1. If no additional expression Y is given, this is the quantum entropy

$$-\operatorname{Tr}(X \log(X)).$$

2. If an additional affine expression Y of same shape as X is given, this is the negative quantum relative entropy

$$\operatorname{Tr}(X \log(Y) - X \log(X))$$

3. If an additional scalar valued real affine expression Y is given, this is the homogenized quantum entropy

$$-\operatorname{Tr}(X \log(X/y))$$

> **Warning:** When you pose a lower bound on this expression, then PICOS enforces $X \succeq 0$ through an auxiliary constraint during solution search. When an additional expression Y is given, PICOS enforces $Y \succeq 0$ as well.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*X*, *Y=None*)

Construct an *`QuantumEntropy`*.

Parameters

- **X** (`AffineExpression`) – The affine expression X.
- **Y** (`AffineExpression`) – An additional affine expression Y. If necessary, PICOS will attempt to reshape or broadcast it to the shape of X.

`__neg__`()

Denote the negation of the expression.

`property X`

The expression X.

`property Y`

The additional expression Y, or `None`.

`property iscomplex`

Whether *`X`* and *`Y`* are complex expressions or not.

`property n`

Length of *`X`*.

picos.expressions.exp_quantkeydist

Implements *`QuantumKeyDistribution`*.

Classes

class `picos.expressions.exp_quantkeydist.`**`QuantumKeyDistribution`**(*X*, *subsystems=0*, *dimensions=2*, *K_list=None*)

Bases: *`Expression`*

Slice of quantum relative entropy used to compute quantum key rates.

Definition

Let X be an $n \times n$-dimensional symmetric or hermitian matrix. Let $\mathcal{Z}$ be the pinching map which maps off-diagonal blocks of a given block structure to zero, i.e., for a bipartite state

$$\mathcal{Z}(X) = \sum_i Z_i X Z_i^\dagger,$$

where $Z_i = |i\rangle\langle i| \otimes \mathbb{I}$ if we block- diagonalize over the first subsystem, and $Z_i = \mathbb{I} \otimes |i\rangle\langle i|$ if we block-diagonalize over the second subsystem. We also generalize this definition to multipartite systems where we block- diagonalize over any number of subsystems.

1. In general, this is the expression

$$-S(\mathcal{G}(X)) + S(\mathcal{Z}(\mathcal{G}(X))),$$

where $S(X) = -\operatorname{Tr}(X \log(X))$ is the quantum entropy, $mathcalG$ is a positive linear map given by Kraus operators

$$\mathcal{G}(X) = \sum_i K_i X K_i^\dagger.$$

2. If `K_list=None`, then $\mathcal{G}$ is assumed to be the identity map, and then this expression is simplified to

$$-S(X) + S(\mathcal{Z}(X)).$$

Warning: When you pose an upper bound on this expression, then PICOS enforces $X \succeq 0$ through an auxiliary constraint during solution search.

`__init__`(*X*, *subsystems=0*, *dimensions=2*, *K_list=None*)

Construct an *`QuantumKeyDistribution`*.

Parameters

- **`X`** (`AffineExpression`) – The affine expression X.
- **`subsystems`** (*`int or tuple or list`*) – A collection of or a single subystem number, indexed from zero, corresponding to subsystems that will be block- diagonalized over. The value -1 refers to the last subsystem.
- **`dimensions`** (*`int or tuple or list`*) – Either an integer d so that the subsystems are assumed to be all of shape $d \times d$, or a sequence of subsystem shapes where an integer d within the sequence is read as $d \times d$. In any case, the elementwise product over all subsystem shapes must equal the expression's shape.
- **`K_list`** (*`None or list(numpy.ndarray)`*) – A list of Kraus operators representing the linear map $\mathcal{G}$. If `K_list=None`, then $\mathcal{G}$ is defined as the identity map.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`property K_list`

The Kraus operators K_i of $\mathcal{G}$.

`property X`

The expression X.

`property Z_list`

The Kraus operators Z_i of $\mathcal{Z}$.

`property dimensions`

The dimensions of the subsystems of X.

`property iscomplex`

Whether *`X`* is a complex expression or not.

`property n`

Length of *`X`*.

`property subsystems`

The subsystems being block-diagonalized of X.

picos.expressions.exp_specnorm

Implements *`SpectralNorm`*.

Classes

class picos.expressions.exp_specnorm.**SpectralNorm**(*x*)

Bases: *Expression*

The spectral norm of a matrix.

This class can represent the spectral norm of a matrix-affine expression (real- or complex valued). The spectral norm is convex, so we can form expressions of the form `SpectralNorm(X) <= t` which are typically reformulated as LMIs that can be handled by SDP solvers.

Definition

If the normed expression is a matrix X, then its spectral norm is

$$\|X\|_2 = \max\{\|Xu\|_2 : \|u\| \leq 1\} = \sqrt{\lambda_{\max}(XX^*)},$$

where $\lambda_{\max}(\cdot)$ denotes the largest eigenvalue of a matrix, and X^* denotes the adjoint matrix of X (i.e., the transposed matrix X^T if X is real-valued).

Special cases:

- If X is scalar, then $\|X\|_2$ reduces to the the absolute value (or modulus) $|X|$.
- If X is scalar, then $\|X\|_2$ coincides with the Euclidean norm of X.

__init__(*x*)

Construct a *SpectralNorm*.

Parameters

x (`ComplexAffineExpression`) – The affine expression to take the norm of.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

__mul__(*other*)

Denote multiplication with another expression on the right.

__rmul__(*other*)

Denote multiplication with another expression on the left.

property x

Real expression whose norm equals that of the original expression.

picos.expressions.exp_sqnorm

Implements *SquaredNorm*.

Classes

class picos.expressions.exp_sqnorm.**SquaredNorm**(*x*)

Bases: *QuadraticExpression*

A squared Euclidean or Frobenius norm.

This is a lightweight wrapper around *QuadraticExpression* that can handle common constraint formulations more efficiently.

__add__(*other*)

Denote addition from the right hand side.

__init__(*x*)

Create a squared Euclidean or Frobenius norm.

Parameters

x – The (complex) affine expression under the squared norm.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

__mul__(*other*)

Denote scaling from the right hand side.

__rmul__(*other*)

Denote scaling from the left hand side.

__truediv__(*other*)

Denote division by a constant scalar.

property argdim

Number of nonzero elements of the expression under the norm.

property fullroot

Affine expression whose squared norm equals the expression.

Overrides `fullroot` of `QuadraticExpression`.

property is0

Whether the expression is zero.

Overrides `is0` of `QuadraticExpression`.

property is_squared_norm

Always `True` for squared norm instances.

Overrides `is_squared_norm` of `QuadraticExpression`.

picos.expressions.exp_sumexp

Implements `SumExponentials`.

Classes

class picos.expressions.exp_sumexp.**SumExponentials**(*x*, *y=None*)

Bases: `Expression`

Sum of elementwise exponentials of an affine expression.

Definition

Let x be an n-dimensional real affine expression.

1. If no additional expression y is given, this is the sum of elementwise exponentials

$$\sum_{i=1}^{n} \exp(\operatorname{vec}(x)_i).$$

2. If an additional affine expression y of same shape as x is given, this is the sum of elementwise perspectives of exponentials

$$\sum_{i=1}^{n} \operatorname{vec}(y)_i \exp\left(\frac{\operatorname{vec}(x)_i}{\operatorname{vec}(y)_i}\right).$$

Warning: When you pose an upper bound t on a sum of elementwise exponentials, then PICOS enforces $t \geq 0$ through an auxiliary constraint during solution search. When an additional expression y is given, PICOS enforces $y \geq 0$ as well.

`__add__`(*other*)

Denote addition with another expression on the right-hand side.

`__init__`(*x*, *y=None*)

Construct a *`SumExponentials`*.

Parameters

- **x** (`AffineExpression`) – The affine expression x.
- **y** (`AffineExpression`) – An additional affine expression y. If necessary, PICOS will attempt to reshape or broadcast it to the shape of x.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__mul__`(*other*)

Denote scaling from the right hand side.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__rmul__`(*other*)

Denote scaling from the left hand side.

`__truediv__`(*other*)

Denote division by a constant scalar.

`property log`

The logarithm of the expression.

`property n`

Length of `x`.

`property x`

The expression x.

`property y`

The additional expression y, or `None`.

picos.expressions.exp_sumxtr

Implements *`SumExtremes`*.

Classes

`class picos.expressions.exp_sumxtr.`**`SumExtremes`**(*x*, *k*, *largest*, *eigenvalues=False*)

Bases: *`Expression`*

Sum of the k largest or smallest elements or eigenvalues.

Definition

Let $k \in \mathbb{Z}_{\geq 1}$.

1. If x is an n-dimensional real vector or matrix and `eigenvalues == False`, then this is the sum of the $k \leq n$ largest or smallest scalar elements of x, depending on the truth value of `largest`.

   Special cases:

   - If $k = 1$, this is either the largest element $\max_{i=1}^{n} \operatorname{vec}(x)_i$ or the smallest element $\min_{i=1}^{n} \operatorname{vec}(x)_i$ of x.
   - If $k = n$, this is the sum of all elements $\langle x, 1 \rangle$ of x.

2. If X is an $n \times n$ hermitian matrix and `eigenvalues == True`, then this is the sum of the $k \leq n$ largest or smallest eigenvalues of X, depending on the truth value of `largest`. Recall that the eigenvalues of a hermitian matrix are real.

   Special cases:

   - If $k = 1$, this is either the largest eigenvalue $\lambda_{\max}(X)$ or the smallest eigenvalue $\lambda_{\min}(X)$ of X.
   - If $k = n$, this equals the trace $\operatorname{tr}(X)$.

If the given k exceeds the n of either case, then k is silently clipped to n.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*x*, *k*, *largest*, *eigenvalues=False*)

Construct a *`SumExtremes`*.

Parameters

- **`x`** (`ComplexAffineExpression`) – The affine expression to take a sum over.
- **`k`** (*`int`*) – Number of summands.
- **`largest`** (*`bool`*) – Whether to sum over the largest (eigen)values as opposed to the smallest.
- **`eigenvalues`** (*`bool`*) – Whether to sum eigenvalues instead of elements.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__mul__`(*other*)

Denote multiplication with another expression on the right.

`__rmul__`(*other*)

Denote multiplication with another expression on the left.

`property eigenvalues`

Whether the sum concerns eigenvalues as opposed to elements.

`property full`

Whether the sum concerns *all* (eigen)values of the expression.

`property k`

Number of (eigen)values to sum.

`property largest`

Whether the sum concerns largest values as opposed to smallest.

`property x`

The expression under the sum.

picos.expressions.exp_wsum

Implements the `WeightedSum` fallback class.

Classes

class `picos.expressions.exp_wsum.`**`WeightedSum`**(*expressions*, *weights=1*, *opstring=None*)

Bases: `Expression`

A convex or concave weighted sum of scalar expressions.

`__add__`(*other*)

Denote addition with another expression on the right-hand side.

`__ge__`(*other*)

Return a constraint that the expression is lower-bounded.

`__init__`(*expressions*, *weights=1*, *opstring=None*)

Construct a weighted sum of expressions.

Parameters

- **`expressions`** – A collection of scalar expressions.
- **`weights`** – A constant weight vector.
- **`opstring`** (`str`) – Used by PICOS internally when this class is tried as a last fallback to represent the result of an otherwise unsupported product or sum.

`__le__`(*other*)

Return a constraint that the expression is upper-bounded.

`__mul__`(*other*)

Denote multiplication with another expression on the right.

`__neg__`()

Denote the negation of the expression.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__rmul__`(*other*)

Denote multiplication with another expression on the left.

`__rsub__`(*other*)

Denote subtraction of the expression from another expression.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

property expressions

The expressions being summed, without their coefficients.

property weights

The coefficient vector as a PICOS column vector.

picos.expressions.expression

Backend for expression type implementations.

Exceptions

exception picos.expressions.expression.**PredictedFailure**

Bases: `TypeError`

Denotes that comparing two expressions will not form a constraint.

Classes

class picos.expressions.expression.**Expression**(*typeStr*, *symbStr*)

Bases: `Valuable`

Abstract base class for mathematical expressions, including mutables.

For mutables, this is the secondary base class, with `Mutable` or a subclass thereof being the primary one.

__abs__()

Denote the default norm of the expression.

The norm used depends on the expression's domain. It is

1. the absolute value of a real scalar,
2. the modulus of a complex scalar,
3. the Euclidean norm of a vector, and
4. the Frobenius norm of a matrix.

__add__(*other*)

Denote addition with another expression on the right-hand side.

__and__(*other*)

Denote horizontal stacking with another expression on the right.

__eq__(*exp*)

Return an equality constraint concerning the expression.

__floordiv__(*other*)

Denote vertical stacking with another expression below.

__ge__(*other*)

Return a constraint that the expression is lower-bounded.

__init__(*typeStr*, *symbStr*)

Perform basic initialization for `Expression` instances.

Parameters

- **typeStr** (`str`) – Short string denoting the expression type.
- **symbStr** (`str`) – Algebraic string description of the expression.

__le__(*other*)

Return a constraint that the expression is upper-bounded.

`__lshift__`(*other*)

Denote either set membership or a linear matrix inequality.

If the other operand is a set, then this denotes that the expression shall be constrained to that set. Otherwise, it is expected that both expressions are square matrices of same shape and this denotes that the expression is upper-bounded by the other expression with respect to the Loewner order (i.e. `other - self` is positive semidefinite).

`__matmul__`(*other*)

Denote the Kronecker product with another expression on the right.

`__mul__`(*other*)

Denote multiplication with another expression on the right.

`__neg__`()

Denote the negation of the expression.

`__or__`(*other*)

Denote the scalar product with another expression on the right.

For (complex) vectors a and b this is the dot product

$$\begin{aligned}(a \mid b) &= \langle a \mid b \rangle \\ &= \langle a, b \rangle \\ &= a \cdot b \\ &= a^H b.\end{aligned}$$

For (complex) matrices A and B this is the Frobenius inner product

$$\begin{aligned}(A \mid B) &= \langle A, B \rangle_F \\ &= A : B \\ &= \operatorname{tr}(A^H B) \\ &= \operatorname{vec}(\overline{A})^T \operatorname{vec}(B)\end{aligned}$$

Note: Write `(A|B)` instead of `A|B` for the scalar product of `A` and `B` to obtain correct operator binding within a larger expression context.

`__pos__`()

Return the expression as-is.

`__pow__`(*other*)

Denote exponentiation with another, scalar expression.

`__radd__`(*other*)

Denote addition with another expression on the left-hand side.

`__rand__`(*other*)

Denote horizontal stacking with another expression on the left.

`__rfloordiv__`(*other*)

Denote vertical stacking with another expression above.

`__rmatmul__`(*other*)

Denote the Kronecker product with another expression on the left.

`__rmul__`(*other*)

Denote multiplication with another expression on the left.

`__ror__`(*other*)

Denote the scalar product with another expression on the left.

See *`__or__`* for details on this operation.

`__rpow__`(*other*)

Denote taking another expression to the power of the expression.

`__rshift__`(*other*)

Denote that the expression is lower-bounded in the Lowener order.

In other words, return a constraint that `self - other` is positive semidefinite.

`__rsub__`(*other*)

Denote subtraction of the expression from another expression.

`__rtruediv__`(*other*)

Denote scalar division of another expression.

`__rxor__`(*other*)

Denote the entrywise product with another expression on the left.

`__sub__`(*other*)

Denote subtraction of another expression from the expression.

`__truediv__`(*other*)

Denote division by another, scalar expression.

`__xor__`(*other*)

Denote the entrywise product with another expression on the right.

`frozen`(*subset=None*)

The expression with valued mutables frozen to their current value.

If all mutables of the expression are valued (and in the subset unless `subset=None`), this is the same as the inversion operation ~.

If the mutables to be frozen do not appear in the expression, then the expression is not copied but returned as is.

Parameters

subset – An iterable of valued *`mutables`* or names thereof that should be frozen. If `None`, then all valued mutables are frozen to their current value. May include mutables that are not present in the expression, but may not include mutables without a value.

Returns Expression

The frozen expression, refined to a more suitable type if possible.

Example

```
>>> from picos import RealVariable
>>> x, y = RealVariable("x"), RealVariable("y")
>>> f = x + y; f
<1×1 Real Linear Expression: x + y>
>>> sorted(f.mutables, key=lambda mtb: mtb.name)
[<1×1 Real Variable: x>, <1×1 Real Variable: y>]
>>> x.value = 5
>>> g = f.frozen(); g  # g is f with x frozen at its current value of 5.
<1×1 Real Affine Expression: [x] + y>
>>> sorted(g.mutables, key=lambda mtb: mtb.name)
[<1×1 Real Variable: y>]
>>> x.value, y.value = 10, 10
>>> f.value  # x takes its new value in f.
```

(continues on next page)

(continued from previous page)

```
20.0
>>> g.value  # x remains frozen at [x] = 5 in g.
15.0
>>> # If an expression is frozen to a constant, this is reversable:
>>> f.frozen().equals(~f) and ~f.frozen() is f
True
```

is_valued()

Whether the expression is valued.

Deprecated since version 2.0: Use `valued` instead.

classmethod make_type(**args*, ***kwargs*)

Create a detailed expression type from subtype parameters.

replace_mutables(*replacement*)

Return a copy of the expression concerning different mutables.

New mutables must have the same shape and vectorization format as the mutables that they replace. This means in particular that `RealVariable`, `IntegerVariable` and `BinaryVariable` of same shape are interchangeable.

If the mutables to be replaced do not appear in the expression, then the expression is not copied but returned as is.

Parameters

replacement (*tuple or list or dict*) – Either a map from mutables or mutable names to new mutables or an iterable of new mutables to replace existing mutables of same name with. See the section on advanced usage for additional options.

Returns Expression

The new expression, refined to a more suitable type if possible.

Advanced replacement

It is also possible to replace mutables with real affine expressions concerning pairwise disjoint sets of fresh mutables. This works only on real-valued mutables that have a trivial internal vectorization format (i.e. `FullVectorization`). The shape of the replacing expression must match the variable's. Additional limitations depending on the type of expression that the replacement is invoked on are possible. The `replacement` argument must be a dictionary.

Example

```
>>> import picos
>>> x = picos.RealVariable("x"); x.value = 1
>>> y = picos.RealVariable("y"); y.value = 10
>>> z = picos.RealVariable("z"); z.value = 100
>>> c = picos.Constant("c", 1000)
>>> a = x + 2*y; a
<1×1 Real Linear Expression: x + 2·y>
>>> a.value
21.0
>>> b = a.replace_mutables({y: z}); b  # Replace y with z.
<1×1 Real Linear Expression: x + 2·z>
>>> b.value
201.0
>>> d = a.replace_mutables({x: 2*x + z, y: c}); d  # Advanced use.
<1×1 Real Affine Expression: 2·x + z + 2·c>
>>> d.value
2102.0
```

`set_value`(*value*)

Set the value of an expression.

Deprecated since version 2.0: Use *`value`* instead.

property `certain`

Always `True` for certain expression types.

This can be `False` for Expression types that inherit from *`UncertainExpression`* (with priority).

property `concave`

Whether the expression is concave.

property `constant`

Whether the expression involves no mutables.

property `convex`

Whether the expression is convex.

property `mutables`

Return the set of mutables that are involved in the expression.

property `parameters`

The set of parameters that are involved in the expression.

property `refined`

A refined version of the expression.

The refined expression can be an instance of a different *`Expression`* subclass than the original expression, if that type is better suited for the mathematical object in question.

The refined expression is automatically used instead of the original one whenever a constraint is created, and in some other places.

The idea behind refined expressions is that operations that produce new expressions can be executed quickly without checking for exceptionnel cases. For instance, the sum of two *`ComplexAffineExpression`* instances could have the complex part eliminated so that storing the result as an *`AffineExpression`* would be prefered, but checking for this case on every addition would be too slow. Refinement is used sparingly to detect such cases at times where it makes the most sense.

Refinement may be disallowed within a context with the *`no_refinement`* context manager. In this case, this property returns the expression as is.

property `scalar`

Whether the expression is scalar.

property `shape`

Return the algebraic shape of the expression.

property `size`

The same as *`shape`*.

property `square`

Whether the expression is a square matrix.

property `string`

Symbolic string representation of the expression.

Use this over Python's `str` if you want to output the symbolic representation even when the expression is valued.

property subtype

The subtype part of the expression's detailed type.

Returns a hashable object that, together with the Python class part of the expression's type, is sufficient to predict the constraint outcome (constraint class and subtype) of any comparison operation with any other expression.

By convention the object returned is a `namedtuple` instance.

property type

The expression's detailed type for constraint prediction.

The returned value is suffcient to predict the detailed type of any constraint that can be created by comparing with another expression.

Since constraints are created from *refined* expressions only, the Python class part of the detailed type may differ from the type of the expression whose *type* is queried.

property uncertain

Always `False` for certain expression types.

This can be `True` for Expression types that inherit from *UncertainExpression* (with priority).

property variables

The set of decision variables that are involved in the expression.

class `picos.expressions.expression.`**`ExpressionType`**(*theClass*, *subtype*)

Bases: *DetailedType*

The detailed type of an expression for predicting constraint outcomes.

This is suffcient to predict the detailed type of any constraint that can be created by comparing with another expression.

predict(*relation*, *other*)

Predict the constraint outcome of comparing expressions.

Parameters

- **relation** – An object from the `operator` namespace representing the operation being predicted.
- **other** (`ExpressionType`) – Another expression type representing the right hand side operand.

Example

```
>>> import operator, picos
>>> a = picos.RealVariable("x") + 1
>>> b = picos.RealVariable("y") + 2
>>> (a <= b).type == a.type.predict(operator.__le__, b.type)
True
```

Functions

`picos.expressions.expression.`**`no_refinement`**()

Context manager that disables the effect of *Expression.refined*.

This can be necessary to ensure that the outcome of a constraint coversion is as predicted, in particular when PICOS uses overridden comparison operators for constraint creation internally.

picos.expressions.expression.**refine_operands**(*stop_at_affine=False*)

Cast `refined` on both operands.

If the left hand side operand (i.e. `self`) is refined to an instance of a different type, then, instead of the decorated method, the method with the same name on the refined type is invoked with the (refined) right hand side operand as its argument.

This decorator is supposed to be used on all constraint creating binary operator methods so that degenerated instances (e.g. a complex affine expression with an imaginary part of zero) can occur but are not used in constraints. This speeds up many computations involving expressions as these degenerate cases do not need to be detected. Note that `Expression.type` also refers to the refined version of an expression.

Parameters

stop_at_affine (`bool`) – Do not refine any affine expressions, in particular do not refine complex affine expressions to real ones.

picos.expressions.expression.**validate_prediction**(*the_operator*)

Validate that the constraint outcome matches the predicted outcome.

picos.expressions.mutable

Implements the `Mutable` base class for variables and parameters.

Classes

class picos.expressions.mutable.**Mutable**(*name*, *vectorization*)

Bases: `ABC`

Primary base class for all variable and parameter types.

Mutables need to inherit this class with priority (first class listed) and the affine expression type that they represent without priority.

__init__(*name*, *vectorization*)

Perform basic initialization for `Mutable` instances.

Parameters

- **name** (`str`) – Name of the mutable. A leading "__" denotes a private mutable and is replaced by a sequence containing the mutable's unique ID.
- **vectorization** (`BaseVectorization`) – Vectorization format used to store the value.

abstract **copy**(*new_name=None*)

Return an independent copy of the mutable.

Note that unlike constraints which keep their ID on copy, mutables are supposed to receive a new id.

id_at(*index*)

Return the unique ID of a scalar entry, assigned at creation.

property **dim**

The mutable's dimension on the real field.

This corresponds to the length of its vectorized value.

property **id**

The unique (starting) ID of the mutable, assigned at creation.

property **internal_value**

The internal (special vectorized) value of the mutable.

property long_string

A string used to represent the mutable in a problem string.

property name

The name of the mutable.

picos.expressions.samples

Implements `Samples`.

Classes

class picos.expressions.samples.**Samples**(*samples=None*, *forced_original_shape=None*, ***kwargs*)

Bases: `object`

A collection of data points.

Example

```
>>> from picos.expressions import Samples
>>> # Load the column-major vectorization of six matrices.
>>> data = [[[1*i, 3*i],
...          [2*i, 4*i]] for i in range(1, 7)]
>>> S = Samples(data)
>>> S
<Samples: (6 4-dimensional samples)>
>>> [S.num, S.dim, S.original_shape]  # Metadata.
[6, 4, (2, 2)]
>>> S.matrix  # All samples as the columns of one matrix.
<4×6 Real Constant: [4×6]>
>>> print(S.matrix)
[ 1.00e+00  2.00e+00  3.00e+00  4.00e+00  5.00e+00  6.00e+00]
[ 2.00e+00  4.00e+00  6.00e+00  8.00e+00  1.00e+01  1.20e+01]
[ 3.00e+00  6.00e+00  9.00e+00  1.20e+01  1.50e+01  1.80e+01]
[ 4.00e+00  8.00e+00  1.20e+01  1.60e+01  2.00e+01  2.40e+01]
>>> print(S[0].T)  # The first sample (transposed for brevity).
[ 1.00e+00  2.00e+00  3.00e+00  4.00e+00]
>>> print(S.mean.T)  # The sample mean (transposed for brevity).
[ 3.50e+00  7.00e+00  1.05e+01  1.40e+01]
>>> print(S.covariance)  # The sample covariance matrix.
[ 3.50e+00  7.00e+00  1.05e+01  1.40e+01]
[ 7.00e+00  1.40e+01  2.10e+01  2.80e+01]
[ 1.05e+01  2.10e+01  3.15e+01  4.20e+01]
[ 1.40e+01  2.80e+01  4.20e+01  5.60e+01]
>>> print(S.original[0])  # The first sample in its original shape.
[ 1.00e+00  3.00e+00]
[ 2.00e+00  4.00e+00]
>>> U = S.select([0, 2, 4])  # Select a subset of samples by indices.
>>> print(U.matrix)
[ 1.00e+00  3.00e+00  5.00e+00]
[ 2.00e+00  6.00e+00  1.00e+01]
[ 3.00e+00  9.00e+00  1.50e+01]
[ 4.00e+00  1.20e+01  2.00e+01]
>>> T, V = S.partition()  # Split into training and validation samples.
>>> print(T.matrix)
[ 1.00e+00  2.00e+00  3.00e+00]
[ 2.00e+00  4.00e+00  6.00e+00]
```

(continues on next page)

(continued from previous page)

```
[ 3.00e+00  6.00e+00  9.00e+00]
[ 4.00e+00  8.00e+00  1.20e+01]
>>> print(V.matrix)
[ 4.00e+00  5.00e+00  6.00e+00]
[ 8.00e+00  1.00e+01  1.20e+01]
[ 1.20e+01  1.50e+01  1.80e+01]
[ 1.60e+01  2.00e+01  2.40e+01]
```

`__init__`(*samples*, *forced_original_shape=None*, *always_copy=True*)

Load a number of data points (samples).

Parameters

- **`samples`** – Any of the following:
  - A tuple or list of constants, each of which denotes a sample vector. Matrices are vectorized but their *`original_shape`* is stored and may be used by PICOS internally.
  - A constant row or column vector whose entries denote scalar samples.
  - A constant matrix whose columns denote the samples.
  - Another *`Samples`* instance. If possible, it is returned as is (*`Samples`* instances are immutable), otherwise a shallow copy with the necessary modifications is returned instead.

  In any case, constants may be given as constant numeric data values (anything recognized by *`load_data`*) or as constant PICOS expressions.
- **`forced_original_shape`** – Overwrites *`original_shape`* with the given shape.
- **`always_copy`** (*bool*) – If this is `False`, then data that is provided in the form of CVXOPT types is not copied but referenced if possible. This can speed up instance creation but will introduce inconsistencies if the original data is modified. Note that this argument has no impact if the `samples` argument already is a *`Samples`* instance; in this case data is never copied.

static `__new__`(*cls*, *samples=None*, *forced_original_shape=None*, ***kwargs*)

Prepare a *`Samples`* instance.

`kfold`(*k*)

Perform k-fold cross-validation (without shuffling).

If random shuffling is desired, write `S.shuffled().kfold(k)` where S is your *`Samples`* instance. To make the shuffling reproducible, see *`shuffled`*.

Returns list(tuple)
A list of k training set and validation set pairs.

Warning: If the number of samples n is not a multiple of k, then the last $n \bmod k$ samples will appear in every training but in no validation set.

Example

```
>>> from picos.expressions import Samples
>>> n, k = 7, 3
>>> S = Samples(range(n))
>>> for i, (T, V) in enumerate(S.kfold(k)):
...     print("Partition {}:\nT = {}V = {}"
...           .format(i + 1, T.matrix, V.matrix))
```

(continues on next page)

(continued from previous page)

```
Partition 1:
T = [ 2.00e+00  3.00e+00  4.00e+00  5.00e+00  6.00e+00]
V = [ 0.00e+00  1.00e+00]

Partition 2:
T = [ 0.00e+00  1.00e+00  4.00e+00  5.00e+00  6.00e+00]
V = [ 2.00e+00  3.00e+00]

Partition 3:
T = [ 0.00e+00  1.00e+00  2.00e+00  3.00e+00  6.00e+00]
V = [ 4.00e+00  5.00e+00]
```

partition(*after_or_fraction=0.5*)

Split the samples into two parts.

Parameters

after_or_fraction (*int or float*) – Either a fraction strictly between zero and one that denotes the relative size of the first partition or an integer that denotes the number of samples to put in the first partition.

select(*indices*)

Return a new Samples instance with only selected samples.

Parameters

indices – The indices of the samples to select.

shuffled(*rng=None*)

Return a randomly shuffled instance of the samples.

Parameters

rng – DEPRECATED

Example

```
>>> from picos.expressions import Samples
>>> S = Samples(range(6))
>>> print(S.matrix)
[ 0.00e+00  1.00e+00  2.00e+00  3.00e+00  4.00e+00  5.00e+00]
>>> import random
>>> random.seed(42)  # Fix seed for reproducibility.
>>> print(S.shuffled().matrix)
[ 3.00e+00  1.00e+00  2.00e+00  4.00e+00  0.00e+00  5.00e+00]
```

property covariance

The sample covariance matrix.

property dim

Sample dimension.

property matrix

A matrix whose columns are the samples.

property mean

The sample mean as a column vector.

property num

Number of samples.

property original

A tuple containing the samples in their original shape.

property original_shape

Original shape of the samples before vectorization.

property vectors

A `tuple` containing the samples as column vectors.

picos.expressions.set

Backend for mathematical set type implementations.

Classes

class picos.expressions.set.**Set**(*typeStr*, *symbStr*)

Bases: `ABC`

Abstract base class for mathematical set expressions.

__init__(*typeStr*, *symbStr*)

Perform basic initialization for `Set` instances.

Parameters

- **typeStr** (*str*) – Short string denoting the set type.
- **symbStr** (*str*) – Algebraic string description of the set.

__lshift__(*other*)

__rshift__(*other*)

classmethod make_type(**args*, ***kwargs*)

Analog to `expression.Expression.make_type`.

replace_mutables(*new_mutables*)

See `replace_mutables`.

property mutables

Return a Python set of mutables that are involved in the set.

property parameters

The set of parameters that are involved in the set.

property refined

The set itself, as sets do not support refinement.

This exists for compatibility with expressions.

property string

Symbolic string representation of the set.

property subtype

Analog to `expression.Expression.subtype`.

property type

Analog to `expression.Expression.type`.

property variables

The set of decision variables that are involved in the set.

class picos.expressions.set.**SetType**(*theClass*, *subtype*)

Bases: `ExpressionType`

`ExpressionType` for sets.

picos.expressions.set_ball

Implements `Ball`.

Classes

class `picos.expressions.set_ball.Ball`(*radius=Constant(1)*, *p=2*, *denominator_limit=1000*)

Bases: `Set`

A ball of radius r according to a (generalized) p-norm.

Definition

In the following, $\|\cdot\|_p$ refers to the vector p-norm or to the entrywise matrix p-norm, depending on the argument. See `Norm` for definitions.

Let $r \in \mathbb{R}$.

1. For $p \in [1, \infty)$ or $p = \infty$ (input as `float("inf")`), this is the convex set

$$\{x \in \mathbb{K} \mid \|x\|_p \leq r\}$$

for any

$$\mathbb{K} \in \bigcup_{m,n \in \mathbb{Z}_{\geq 1}} \left(\mathbb{C}^n \cup \mathbb{C}^{m \times n}\right).$$

2. For a generalized p-norm with $p \in (0, 1)$, this is the convex set

$$\{x \in \mathbb{K} \mid \|x\|_p \geq r \wedge x \geq 0\}$$

for any

$$\mathbb{K} \in \bigcup_{m,n \in \mathbb{Z}_{\geq 1}} \left(\mathbb{R}^n \cup \mathbb{R}^{m \times n}\right).$$

Note that x may not be complex if $p < 1$ due to the implicit $x \geq 0$ constraint in this case, which is not meaningful on the complex field.

Note further that r may be any scalar affine expression, it does not need to be constant.

Note: Due to significant differences in scope, `Ball` is not a subclass of `Ellipsoid` even though both classes can represent Euclidean balls around the origin.

`__init__`(*radius=Constant(1)*, *p=2*, *denominator_limit=1000*)

Construct a p-norm ball of given radius.

Parameters

- **radius** (*`float` or `AffineExpression`*) – The ball's radius.
- **p** (*`float`*) – The value for p, which is cast to a limited precision fraction.
- **denominator_limit** (*`int`*) – The largest allowed denominator when casting p to a fraction. Higher values can yield a greater precision at reduced performance.

property p

The value p defining the p-norm used.

This is a limited precision version of the parameter used when the ball was constructed.

property r

The ball's radius r.

picos.expressions.set_ellipsoid

Implements *Ellipsoid*.

Classes

class picos.expressions.set_ellipsoid.**Ellipsoid**(*n*, *A='I'*, *c=0*)

Bases: *Set*

An affine transformation of the Euclidean unit ball.

Definition

For $n \in \mathbb{Z}_{\geq 1}$, $A \in \mathbb{R}^{n \times n}$ invertible and $c \in \mathbb{R}^n$, an instance of this class represents the set

$$\begin{aligned} &\{Ax + c \mid \|x\|_2 \leq 1\} \\ =&\{x \mid \|A^{-1}(x - c)\|_2 \leq 1\} \\ =&\{x \mid \|x - c\|_{(A^{-1})^T A^{-1}} \leq 1\}. \end{aligned}$$

Unlike most sets, instances of this class offer a limited set of algebraic operations that are generalized from expressions to sets in the natural way. In particular, you can add or substract constant vectors of matching dimension and apply matrix multiplication from the left hand side, both of which will act on the term $Ax + c$ in the definition above.

Example

```
>>> from picos import Ellipsoid, RealVariable
>>> Ellipsoid(3)  # Three-dimensional Euclidean unit ball.
<Centered Unit Ball: {I·x : ‖x‖ ≤ 1}>
>>> Ellipsoid(3, range(9))  # Linear transformation of the unit ball.
<Centered Ellipsoid: {[3×3]·x : ‖x‖ ≤ 1}>
>>> Ellipsoid(3, "2I", 1)  # Offset ball of radius two.
<Offset Ellipsoid: {2·I·x + [1] : ‖x‖ ≤ 1}>
>>> 2*Ellipsoid(3) + 1  # The same using algebraic operations.
<Offset Ellipsoid: {2·I·x + [1] : ‖x‖ ≤ 1}>
>>> x = RealVariable("x", 3)
>>> (2*x + range(3)) << (4*Ellipsoid(3) + 5)  # Constraint creation.
<4×1 SOC Constraint: ‖(4·I)^(-1)·(2·x + [3×1] - [5])‖ ≤ 1>
```

Note: Due to significant differences in scope, *Ellipsoid* is not a superclass of *Ball* even though both classes can represent Euclidean balls around the origin.

__add__(*other*)

__init__(*n*, *A='I'*, *c=0*)

Construct an ellipsoid.

Parameters

- **n** (*int*) – Dimensionality n of the ellipsoid.
- **A** (*AffineExpression* or recognized by *load_data*) – Invertible linear transformation matrix A.
- **c** (*AffineExpression* or recognized by *load_data*) – The center c of the ellispoid.

Warning: Invertibility of A is not checked on instanciation. If A is singular, a `RuntimeError` is raised once the inverse is needed.

__mul__(*other*)

__radd__(*other*)

__rmul__(*other*)

__truediv__(*other*)

property A
The linear operator matrix A.

property Ainv
The inverse linear operator matrix A^{-1}.

property c
The center point c.

property dim
The dimensionality n.

picos.expressions.set_simplex

Implements `Simplex`.

Classes

`class picos.expressions.set_simplex.`**`Simplex`**(*radius=Constant(1)*, *truncated=False*, *symmetrized=False*)

Bases: `Set`

A (truncated, symmetrized) real simplex.

Definition

Let $r \in \mathbb{R}_{\geq 0}$ the specified radius and $n \in \mathbb{Z}_{\geq 1}$ an arbitrary dimensionality.

1. Without truncation and symmetrization, this is the nonnegative simplex

$$\{x \in \mathbb{R}^n_{\geq 0} \mid \sum_{i=1}^n x_i \leq r\}.$$

For $r = 1$, this is the standard (unit) n-simplex.

2. With truncation but without symmetrization, this is the nonnegative simplex intersected with the ∞-norm unit ball

$$\{x \in \mathbb{R}^n_{\geq 0} \mid \sum_{i=1}^n x_i \leq r \wedge x \leq 1\}.$$

For $r \leq 1$, this equals case (1).

3. With symmetrization but without truncation, this is the 1-norm ball of radius r

$$\{x \in \mathbb{R}^n \mid \sum_{i=1}^n |x_i| \leq r\}.$$

4. With both symmetrization and truncation, this is the convex polytope

$$\{x \in \mathbb{R} \mid \sum_{i=1}^n |x_i| \leq r \wedge 0 \leq x \leq 1\}.$$

For $r \leq 1$, this equals case (3).

__init__(*radius=Constant(1)*, *truncated=False*, *symmetrized=False*)

Construct a `Simplex`.

Parameters

radius (`float` *or* `AffineExpression`) – The radius of the simplex.

property radius

The radius of the simplex.

property symmetrized

Wether the simplex is mirrored onto all orthants.

property truncated

Whether this is intersected with the unit ∞-ball.

picos.expressions.uncertain

Expression types with an explicit representation of data uncertainty.

Exceptions

exception `picos.expressions.uncertain.`**`IntractableWorstCase`**

See *`picos.expressions.uncertain.uexpression.IntractableWorstCase`*.

Classes

class `picos.expressions.uncertain.`**`ConicPerturbationSet`**

See *`picos.expressions.uncertain.pert_conic.ConicPerturbationSet`*.

class `picos.expressions.uncertain.`**`MomentAmbiguitySet`**

See *`picos.expressions.uncertain.pert_moment.MomentAmbiguitySet`*.

class `picos.expressions.uncertain.`**`Perturbation`**

See *`picos.expressions.uncertain.perturbation.Perturbation`*.

class `picos.expressions.uncertain.`**`PerturbationUniverse`**

See *`picos.expressions.uncertain.perturbation.PerturbationUniverse`*.

class `picos.expressions.uncertain.`**`RandomExtremumAffine`**

See *`picos.expressions.uncertain.uexp_rand_pwl.RandomExtremumAffine`*.

class `picos.expressions.uncertain.`**`RandomMaximumAffine`**

See *`picos.expressions.uncertain.uexp_rand_pwl.RandomMaximumAffine`*.

class `picos.expressions.uncertain.`**`RandomMinimumAffine`**

See *`picos.expressions.uncertain.uexp_rand_pwl.RandomMinimumAffine`*.

class `picos.expressions.uncertain.`**`ScenarioPerturbationSet`**

See *`picos.expressions.uncertain.pert_scenario.ScenarioPerturbationSet`*.

class `picos.expressions.uncertain.`**`UncertainAffineExpression`**

See *`picos.expressions.uncertain.uexp_affine.UncertainAffineExpression`*.

class `picos.expressions.uncertain.`**`UncertainExpression`**

See *`picos.expressions.uncertain.uexpression.UncertainExpression`*.

class `picos.expressions.uncertain.`**`UncertainNorm`**

See *`picos.expressions.uncertain.uexp_norm.UncertainNorm`*.

class picos.expressions.uncertain.**UncertainSquaredNorm**

See *picos.expressions.uncertain.uexp_sqnorm.UncertainSquaredNorm*.

class picos.expressions.uncertain.**UnitBallPerturbationSet**

See *picos.expressions.uncertain.pert_conic.UnitBallPerturbationSet*.

class picos.expressions.uncertain.**WassersteinAmbiguitySet**

See *picos.expressions.uncertain.pert_wasserstein.WassersteinAmbiguitySet*.

picos.expressions.uncertain.pert_conic

Implements *ConicPerturbationSet*.

Classes

class picos.expressions.uncertain.pert_conic.**ConicPerturbationSet**(*parameter_name*, *shape=(1, 1)*)

Bases: *PerturbationUniverse*

A conic description of a *Perturbation*.

Definition

An instance Θ of this class defines a perturbation parameter

$$\theta \in \Theta = \{t \in \mathbb{R}^{m \times n} \mid \exists u \in \mathbb{R}^l : A \operatorname{vec}(t) + Bu + c \in K\}$$

where $m, n \in \mathbb{Z}_{\geq 1}$, $l \in \mathbb{Z}_{\geq 0}$, $A \in \mathbb{R}^{k \times mn}$, $B \in \mathbb{R}^{k \times l}$, $c \in \mathbb{R}^k$ and $K \subseteq \mathbb{R}^k$ is a (product) cone for some $k \in \mathbb{Z}_{\geq 1}$.

Usage

Obtaining θ is done in a number of steps:

1. Create an instance of this class (see *__init__*).
2. Access *element* to obtain a regular, fresh *RealVariable* representing t.
3. Define Θ through any number of regular PICOS constraints that depend only on t and that have a conic representation by passing the constraints to *bound*.
4. Call *compile* to obtain a handle to the *Perturbation* θ.
5. You can now use θ to build instances of *UncertainAffineExpression* and derived constraint types.

It is best practice to assign t to a Python variable and overwrite that variable with θ on compilation.

Alternatively, you can obtain a compiled Θ from the factory method *from_constraints* and access θ via *parameter*.

Example

```
>>> from picos import Constant, Norm, RealVariable
>>> from picos.uncertain import ConicPerturbationSet
>>> S = ConicPerturbationSet("P", (4, 4))
>>> P = S.element; P  # Obtain a temporary parameter to describe S with.
<4×4 Real Variable: P>
>>> S.bound(Norm(P, float("inf")) <= 1)  # Confine each element to [-1,1].
>>> S.bound(Norm(P, 1) <= 4); S  # Allow a perturbation budget of 4.
<4×4 Conic Perturbation Set: {P : ‖P‖_max ≤  1 ∨ ‖P‖_sum ≤  4}>
>>> P = S.compile(); P  # Compile the set and obtain the actual parameter.
<4×4 Perturbation: P>
```

(continues on next page)

(continued from previous page)

```
>>> A = Constant("A", range(16), (4, 4))
>>> U = A + P; U  # Define an uncertain data matrix.
<4×4 Uncertain Affine Expression: A + P>
>>> x = RealVariable("x", 4)
>>> U*x  # Define an uncertain affine expression.
<4×1 Uncertain Affine Expression: (A + P)·x>
```

__init__(*parameter_name*, *shape=(1, 1)*)

Create a ConicPerturbationSet.

Parameters

- **parameter_name** (*str*) – Name of the parameter that lives in the set.
- **shape** (*int or tuple or list*) – The shape of a vector or matrix perturbation.

bound(*constraint*)

Add a constraint that bounds t.

The constraints do not need to be conic but they need to have a *conic representation*, which may involve any number of auxiliary variables. For instance, given a constant *uncertainty budget* b, you may add the bound $\|t\|_1 \leq b$ (via `picos.Norm(t, 1) <= b`) which can be represented in conic form as

$$
\begin{aligned}
&\exists v \in \mathbb{R}^{\dim(t)} : -t \leq v \wedge t \leq v \wedge \mathbf{1}^T v \leq b \\
\iff &\exists v \in \mathbb{R}^{\dim(t)} : \begin{pmatrix} v+t \\ v-t \\ b-\mathbf{1}^T v \end{pmatrix} \in \mathbb{R}^{2\dim(t)+1}_{\geq 0}.
\end{aligned}
$$

The auxiliary variable v then plays the role of (a slice of) u in the formal definition of Θ.

When you are done adding bounds, you can obtain θ using compile.

Raises

RuntimeError – If the set was already compiled.

compile(*validate_feasibility=False*)

Compile the set and return θ.

Internally, this computes the matrices A and B, the vector c and the (product) cone K.

Parameters

validate_feasibility (*bool*) – Whether to solve the feasibility problem associated with the bounds on t to verify that Θ is nonempty.

Returns

An instance of Perturbation.

Raises

- **RuntimeError** – If the set was already compiled.
- **TypeError** – If the bound constraints could not be put into conic form.
- **ValueError** – If Θ could not be verified to be nonempty (needs `validate_feasibility=True`).

classmethod from_constraints(*parameter_name*, **constraints*)

Create a ConicPerturbationSet from constraints.

The constraints must concern a single regular decision variable that plays the role of the element t. This variable is not stored or modified and can be reused in a different context.

Parameters

- **parameter_name** (*str*) – Name of the parameter that lives in the set.

- **constraints** – A parameter sequence of constraints that concern a single regular decision variable whose internal vectorization is trivial (its dimension must match the product over its shape) and that have a conic representation.

Raises

- `ValueError` – If the constraints do not all concern the same single variable.
- `TypeError` – If the variable uses a nontrivial vectorization format or if the constraints do not all have a conic representation.

Example

```
>>> from picos.expressions.uncertain import ConicPerturbationSet
>>> from picos import RealVariable
>>> x = RealVariable("x", 4)
>>> T = ConicPerturbationSet.from_constraints("t", abs(x) <= 2, x >= 0)
>>> print(T)
{t : ‖t‖ ≤  2 ∨ t ≥  0}
>>> print(repr(T.parameter))
<4×1 Perturbation: t>
```

worst_case(*scalar*, *direction*)

Implement for *PerturbationUniverse*.

property A

The compiled matrix A.

Raises
`RuntimeError` – If the set has not been compiled.

property B

The compiled matrix B or `None` if $l = 0$.

Raises
`RuntimeError` – If the set has not been compiled.

property K

The compiled (product) cone K.

Raises
`RuntimeError` – If the set has not been compiled.

property c

The compiled vector c.

Raises
`RuntimeError` – If the set has not been compiled.

property distributional

Implement for *PerturbationUniverse*.

property element

The perturbation element t describing the set.

This is a regular *RealVariable* that you can use to create constraints to pass to *bound*. You can then obtain the "actual" perturbation parameter θ to use in expressions alongside your decision variaiables using *compile*.

> **Warning:** If you use this object instead of *parameter* to define a decision problem then it will act as a regular decision variable, which is probably not what you want.

Raises

`RuntimeError` – If the set was already compiled.

property ellipsoidal

Whether the perturbation set is an ellipsoid.

If this is true, then a *unit_ball_form* is available.

property parameter

The perturbation parameter θ living in the set.

This is the object returned by *compile*.

Raises

`RuntimeError` – If the set has not been compiled.

property unit_ball_form

A recipe to repose from ellipsoidal to unit norm ball uncertainty.

If the set is *ellipsoidal*, then this is a pair `(U, M)` where `U` is a *UnitBallPerturbationSet* and `M` is a dictionary mapping the old *parameter* to an affine expression of the new parameter that can represent the old parameter in an expression (see *replace_mutables*). The mapping `M` is empty if and only if the perturbation set is already an instance of *UnitBallPerturbationSet*.

If the uncertainty set is not ellipsoidal, then this is `None`.

See also *SOCConstraint.unit_ball_form*.

Example

```
>>> from picos import Problem, RealVariable, sum
>>> from picos.uncertain import ConicPerturbationSet
>>> # Create a conic perturbation set and a refinement recipe.
>>> T = ConicPerturbationSet("t", (2, 2))
>>> T.bound(abs(([[1, 2], [3, 4]] ^ T.element) + 1) <= 10)
>>> t = T.compile()
>>> U, mapping = T.unit_ball_form
>>> print(U)
{t' : ‖t'‖ ≤  1}
>>> print(mapping)
{<2×2 Perturbation: t>: <2×2 Uncertain Affine Expression: t(t')>}
>>> # Define and solve a conically uncertain LP.
>>> X = RealVariable("X", (2, 2))
>>> P = Problem()
>>> P.set_objective("max", sum(X))
>>> _ = P.add_constraint(X + 2*t <= 10)
>>> print(repr(P.parameters["t"].universe))
<2×2 Conic Perturbation Set: {t : ‖[2×2]⊙t + [1]‖ ≤  10}>
>>> _ = P.solve(solver="cvxopt")
>>> print(X)
[-8.00e+00  1.00e+00]
[ 4.00e+00  5.50e+00]
>>> # Refine the problem to a unit ball uncertain LP.
>>> Q = Problem()
>>> Q.set_objective("max", sum(X))
>>> _ = Q.add_constraint(X + 2*mapping[t] <= 10)
>>> print(repr(Q.parameters["t'"].universe))
<2×2 Unit Ball Perturbation Set: {t' : ‖t'‖ ≤  1}>
>>> _ = Q.solve(solver="cvxopt")
>>> print(X)
[-8.00e+00  1.00e+00]
[ 4.00e+00  5.50e+00]
```

class picos.expressions.uncertain.pert_conic.**UnitBallPerturbationSet**(*parameter_name*, *shape=(1, 1)*)

Bases: `ConicPerturbationSet`

Represents perturbation in an Euclidean or Frobenius unit norm ball.

This is a `ConicPerturbationSet` with fixed form

$$\{t \in \mathbb{R}^{m \times n} \mid \|t\|_F \leq 1\}.$$

After initialization, you can obtain the parameter using `parameter`.

__init__(*parameter_name*, *shape=(1, 1)*)

See `ConicPerturbationSet.__init__`.

property unit_ball_form

Overwrite `ConicPerturbationSet.unit_ball_form`.

picos.expressions.uncertain.pert_moment

Implements `MomentAmbiguitySet`.

Classes

class picos.expressions.uncertain.pert_moment.**MomentAmbiguitySet**(*parameter_name*, *shape*, *nominal_mean=0*, *nominal_covariance='I'*, *alpha=0*, *beta=1*, *sample_space=None*)

Bases: `PerturbationUniverse`

A moment-uncertain description of a random perturbation parameter.

Model of uncertainty

As a distributional ambiguity set, an instance $\mathcal{P}$ of this class

1. represents a safety region for a partially known (ambiguous) probability distribution $\Xi \in \mathcal{P}$ and
2. provides a random, ambiguously distributed perturbation parameter $\xi \sim \Xi$ that can be used to define worst-case-expectation expressions of the form

$$\underset{\Xi \in \mathcal{P}}{(\max\ or\ \min)} \mathbb{E}_\Xi[f(x, \xi)]$$

for a selection of functions f and a decision variable x.

Definition

Formally, this class can describe ambiguity sets of the form

$$\mathcal{P} = \left\{ \Xi \in \mathcal{M} \,\middle|\, \begin{aligned} \mathbb{P}(\xi \in \mathcal{S}) = 1, \\ (\mathbb{E}[\xi] - \mu)^T \Sigma^{-1} (\mathbb{E}[\xi] - \mu) \leq \alpha, \\ \mathbb{E}[(\xi - \mu)(\xi - \mu)^T] \preceq \beta\Sigma \end{aligned} \right\}$$

where

1. $\mathcal{M}$ is the set of all Borel probability measures on $\mathbb{R}^n$,
2. the sample space $\mathcal{S} \subseteq \mathbb{R}^n$ bounds the support of Ξ and may be given as either $\mathbb{R}^n$ or as an n-dimensional ellipsoid,

3. $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{S}^n$ with $\Sigma \succ 0$ are the **nominal** mean and covariance matrix of Ξ, respectively, and
4. $\alpha \geq 0$ and $\beta \geq 1$ are meta-parameters bounding the uncertainty concerning the mean and the covariance matrix, respectively.

Unless $\mathcal{S} = \mathbb{R}^n$, this class can also represent the limit cases of $\alpha \to \infty$ and $\beta \to \infty$ denoting an unbounded mean and covariance matrix, respectively.

Note: While $\alpha = 0$ denotes that the nominal mean μ is certain, there is a subtle difference between setting $\beta = 1$ on the one hand and assuming a certain form for the covariance matrix on the other hand: In the former case, the worst case covariance matrix may be Lowener smaller than the nominal one. Setting a lower bound on the covarianve matrix is computationally difficult and not supported.

Supported functions

1. A squared norm $f(x, \xi) = \|A(x, \xi)\|_F^2$ where A is biaffine in x and ξ. This can be written as `abs(A)**2` in Python.
2. A convex piecewise linear function $f(x, \xi) = max_{i=1}^k a_i(x, \xi)$ where a is biaffine in x and ξ for all $i \in [k]$. This can be written as `picos.max([a_1, ..., a_k])` in Python.
3. A concave piecewise linear function $f(x, \xi) = min_{i=1}^k a_i(x, \xi)$ where a is biaffine in x and ξ for all $i \in [k]$. This can be written as `picos.min([a_1, ..., a_k])` in Python.

Example

We show that for unbounded mean and support (i.e. $\alpha \to \infty$ and $\beta \to \infty$) and for a finite samples space S, this distributional ambiguity set can be used in the context of classical (non-distributional) robust optimization applied to least squares problems.

```
>>> from picos import Constant, diag, Ellipsoid, Problem, RealVariable
>>> from picos.uncertain import ConicPerturbationSet, MomentAmbiguitySet
>>> import numpy
>>> numpy.random.seed(1)
>>> # Setup data and variables of the nominal least squares problem.
>>> n = 3
>>> A = Constant("A", numpy.random.random((n, n)))
>>> b = Constant("b", numpy.random.random(n))
>>> x = RealVariable("x", n)
>>> # Setup an ellipsoid S bounding the uncertainty in both models.
>>> N = n**2
>>> S = Ellipsoid(N, diag(range(1, N + 1)), range(N))
>>> # Define and solve both robust models.
>>> U1 = ConicPerturbationSet.from_constraints(
...     "Y", RealVariable("Y", N) << S)
>>> U2 = MomentAmbiguitySet("Z", N, alpha=None, beta=None, sample_space=S)
>>> Y = U1.parameter.reshaped((n, n))
>>> Z = U2.parameter.reshaped((n, n))
>>> P1, P2 = Problem(), Problem()
>>> P1.objective = "min", abs((A + Y)*x - b)
>>> P2.objective = "min", abs((A + Z)*x - b)**2
>>> _ = P1.solve(solver="cvxopt")
>>> x1 = ~x  # Save current value of x as a constant PICOS expression x1.
>>> _ = P2.solve(solver="cvxopt")
>>> x2 = ~x
>>> # Verify that both models yield the same robust regression vector.
>>> x1.equals(x2, relTol=1e-4)
True
```

`__init__`(*parameter_name*, *shape*, *nominal_mean=0*, *nominal_covariance='I'*, *alpha=0*, *beta=1*, *sample_space=None*)

Create a *`MomentAmbiguitySet`*.

Parameters

- **`parameter_name`** (*`str`*) – Name of the random parameter ξ.
- **`shape`** (anything recognized by *`load_shape`*) – Shape of ξ. Must characterize a column vector. If `None`, then the shape is inferred from the nominal mean.
- **`nominal_mean`** (anything recognized by *`load_data`*) – The nominal mean μ of the ambiguous distribution Ξ.
- **`nominal_covariance`** (anything recognized by *`load_data`*) – The nominal covariance matrix Σ of the ambiguous distribution Ξ.
- **`alpha`** (*`float`*) – The parameter $\alpha \geq 0$ bounding the uncertain mean. The values `None` and `float("inf")` denote an unbounded mean.
- **`beta`** (*`float`*) – The parameter $\beta \geq 1$ bounding the uncertain covariance matrix. The values `None` and `float("inf")` denote unbounded covariances.
- **`sample_space`** (`None` or *`TheField`* or *`Ellipsoid`*) – The sample space $\mathcal{S}$. If this is `None` or an instance of *`TheField`* (i.e. $\mathbb{R}^n$), then the support of Ξ is unbounded. If this is an n-dimensional instance of *`Ellipsoid`*, then the support of Ξ is a subset of that ellipsoid.

`property alpha`

The parameter α.

A value of `None` denotes $\alpha \to \infty$.

`property beta`

The parameter β.

A value of `None` denotes $\beta \to \infty$.

`property dim`

The dimension n of the sample space.

`property distributional`

Implement for *`PerturbationUniverse`*.

`property nominal_covariance`

The nominal covariance matrix Σ.

`property nominal_mean`

The nominal mean μ of the ambiguous distribution.

`property parameter`

The random perturbation parameter ξ.

`property sample_space`

The sample space (bound on support) $\mathcal{S}$.

A value of `None` means $\mathcal{S} = \mathbb{R}^n$.

picos.expressions.uncertain.pert_scenario

Implements *ScenarioPerturbationSet*.

Classes

class picos.expressions.uncertain.pert_scenario.**ScenarioPerturbationSet**(*parameter_name*, *scenarios*, *compute_hull=None*)

Bases: *PerturbationUniverse*

A scenario description of a *Perturbation*.

Definition

An instance Θ of this class defines a perturbation parameter

$$\theta \in \Theta = \operatorname{conv}(S)$$

where $S \subset \mathbb{R}^{m \times n}$ is a finite set of *scenarios* and conv denotes the convex hull.

Usually, the scenarios are observed or projected realizations of the uncertain data and θ is used to represent the data directly.

Example

```
>>> from picos.uncertain import ScenarioPerturbationSet
>>> scenarios = [[1, -1], [1, 1], [-1, -1], [-1, 1], [0, 0]]
>>> S = ScenarioPerturbationSet("s", scenarios, False); S
<2×1 Scenario Perturbation Set: conv({5 2×1 scenarios})>
>>> s = S.parameter; s
<2×1 Perturbation: s>
>>> # Compute largest sum of entries over all points in S.
>>> value, realization = S.worst_case(s[0] + s[1], "max")
>>> round(value,  4)
2.0
>>> print(realization)
[ 1.00e+00]
[ 1.00e+00]
```

__init__(*parameter_name*, *scenarios*, *compute_hull=None*)

Create a *ScenarioPerturbationSet*.

Parameters

- **parameter_name** (*str*) – Name of the parameter that lives in the set.
- **scenarios** (anything recognized by *picos.Samples*) – A collection of data points of same shape representing S.
- **compute_hull** (*bool*) – Whether to use SciPy to compute the convex hull of the data points and discard points in the interior. This can speed up the solution process significantly, in particular when the scenarios come from observations and when the data is low-dimensional. On the other hand, when the given scenarios are known to be on the boundary of their convex hull, then disabling this speeds up initialization of the perturbation set. The default value of `None` means `True` when SciPy is available and `False` otherwise.

worst_case(*scalar*, *direction*)

Implement for *PerturbationUniverse*.

property distributional

Implement for *PerturbationUniverse*.

property parameter

Implement for *PerturbationUniverse*.

property scenarios

The registered scenarios as a *Samples* object.

picos.expressions.uncertain.pert_wasserstein

Implements *WassersteinAmbiguitySet*.

Classes

class picos.expressions.uncertain.pert_wasserstein.**WassersteinAmbiguitySet**(*parameter_name*, *p*, *eps*, *samples*, *weights=1*)

Bases: *PerturbationUniverse*

A wasserstein ambiguity set centered at a discrete distribution.

Model of uncertainty

As a distributional ambiguity set, an instance $\mathcal{P}$ of this class

1. represents a safety region for a partially known (ambiguous) probability distribution $\Xi \in \mathcal{P}$ and
2. provides a random, ambiguously distributed perturbation parameter $\xi \sim \Xi$ that can be used to define worst-case-expectation expressions of the form

$$\underset{\Xi \in \mathcal{P}}{(\max\ or\ \min)} \mathbb{E}_\Xi[f(x, \xi)]$$

for a selection of functions f and a decision variable x.

Definition

Formally, this class can describe discrepancy-based ambiguity sets of the form

$$\mathcal{P} = \{\Xi \in \mathcal{M} \mid \mathrm{W}_p(\Xi, \Xi_\mathrm{N}) \leq \epsilon\}$$

where discrepancy from the discrete nominal distribution

$$\Xi_\mathrm{N} = \sum_{i=1}^{N} w_i \delta_{\xi_{(i)}} \in \mathcal{M}$$

is measured with respect to the Wasserstein distance of order $p \geq 1$,

$$\mathrm{W}_p(\Xi, \Xi') = \left(\inf_{\Phi \in \Pi(\Xi, \Xi')} \int_{\mathbb{R}^m \times \mathbb{R}^m} \|\phi - \phi'\|^p \ \Phi(\mathrm{d}\phi \times \mathrm{d}\phi') \right)^{\frac{1}{p}},$$

where

1. $\mathcal{M}$ is the set of all Borel probability measures on $\mathbb{R}^n$ for some $n \in \mathbb{Z}_{\geq 1}$,
2. $\Pi(\Xi, \Xi')$ denotes the set of all couplings of Ξ and Ξ',
3. $\xi_{(i)} \in \mathbb{R}^n$ for all $i \in [N]$ are the $N \in \mathbb{Z}_{\geq 1}$ *samples* comprising the support of Ξ_N,
4. $w_i \in \mathbb{R}_{\geq 0}$ are *weights* denoting the nominal probabilitiy mass at $\xi_{(i)}$ for all $i \in [N]$,

5. $\delta_{\xi_{(i)}}$ denotes the Dirac delta function with unit mass at $\xi_{(i)}$ for all $i \in [N]$ and where
6. $\epsilon \in \mathbb{R}_{\geq 0}$ controls the radius of the ambiguity set.

Supported functions

For $p = 1$:

1. A convex piecewise linear function $f(x, \xi) = max_{i=1}^{k} a_i(x, \xi)$ where a is biaffine in x and ξ for all $i \in [k]$. This can be written as `picos.max([a_1, ..., a_k])` in Python.
2. A concave piecewise linear function $f(x, \xi) = min_{i=1}^{k} a_i(x, \xi)$ where a is biaffine in x and ξ for all $i \in [k]$. This can be written as `picos.min([a_1, ..., a_k])` in Python.

For $p = 2$:

1. A squared norm $f(x, \xi) = \|A(x, \xi)\|_F^2$ where A is biaffine in x and ξ. This can be written as `abs(A)**2` in Python.

`__init__`(*parameter_name*, *p*, *eps*, *samples*, *weights=1*)

Create a `WassersteinAmbiguitySet`.

Parameters

- **`parameter_name`** (*`str`*) – Name of the random parameter ξ.
- **`p`** (*`float`*) – The Wasserstein type/order parameter p.
- **`eps`** (*`float`*) – The Wasserstein ball radius ϵ.
- **`samples`** (aynthing recognized by `Samples`) – The support of the discrete distribution Ξ_D given as the *samples* $\xi_{(i)}$. The original shape of the samples determines the shape of ξ.
- **`weights`** – A vector denoting the nonnegative weight (e.g. frequency or probability) of each sample. Its length must match the number of samples provided. The argument will be normalized such that its entries sum to one. Entries of zero will be dropped alongside their associated sample. The default value of 1 denotes the empirical distribution on the samples.

Warning: Duplicate samples are not detected and can impact performance. If duplicate samples are likely, make sure to detect them and encode their frequency in the weight vector.

`property distributional`

Implement for `PerturbationUniverse`.

`property eps`

The Wasserstein ball radius ϵ.

`property p`

The Wasserstein order p.

`property parameter`

The random perturbation parameter ξ.

`property samples`

The registered samples as a `Samples` object.

`property weights`

The sample weights a constant PICOS vector.

picos.expressions.uncertain.perturbation

Implements a parameterization for (random) noise in data.

Classes

`class picos.expressions.uncertain.perturbation.`**`Perturbation`**`(`*universe*, *name*, *shape*`)`

Bases: `Mutable`, `UncertainAffineExpression`

A parameter that can be used to describe (random) noise in data.

This is the initial building block for an `UncertainAffineExpression`. In particular, an affine transformation of this parameter represents uncertain data.

`__init__``(`*universe*, *name*, *shape*`)`

Create a `Perturbation`.

Parameters

- **`universe`** (`PerturbationUniverse`) – Either the set that the perturbation parameter lives in or the distribution according to which the perturbation is distributed.
- **`name`** (`str`) – Symbolic string description of the perturbation, similar to a variable's name.
- **`shape`** (`int or tuple or list`) – Algebraic shape of the perturbation parameter.

This constructor is meant for internal use. As a user, you will want to first define a universe (e.g. `ConicPerturbationSet`) for the parameter and obtain the parameter from it.

`copy``(`*new_name=None*`)`

Return an independent copy of the perturbation.

`property universe`

The uncertainty universe that the parameter belongs to.

`class picos.expressions.uncertain.perturbation.`**`PerturbationUniverse`**

Bases: `ABC`

Base class for uncertain perturbation sets and distributions.

See `distributional` for a distinction between perturbation sets, random distributions and distributional ambiguity sets, all three of which can be represented by this class.

The naming scheme for implementing classes is as follows:

- Perturbation sets (robust optimization) end in `PerturbationSet`,
- random distributions (stochastic programming) end in `Distribution`,
- distributional ambiguity sets (DRO) end in `AmbiguitySet`.

`classmethod make_type``(`**args*, ***kwargs*`)`

Create a detailed universe type from subtype parameters.

`worst_case``(`*scalar*, *direction*`)`

Find a worst-case realization of the uncertainty for an expression.

Parameters

- **`scalar`** (`UncertainExpression`) – A scalar uncertain expression that depends only on the perturbation `parameter`.
- **`direction`** (`str`) – Either `"min"` or `"max"`, denoting the worst-case direction.

Returns

A pair where the first element is the worst-case (expeceted) value as a `float` and where the second element is a realization of the perturbation parameter that attains this worst case as a `float` or CVXOPT matrix (or `None` for stochastic uncertainty).

Raises

- **`TypeError`** – When the function is not scalar.
- **`ValueError`** – When the function depends on other mutables than exactly the *`parameter`*.
- ***`picos.uncertain.IntractableWorstCase`*** – When computing the worst-case (expected) value is not supported, in particular when it would require solving a nonconvex problem.
- **`RuntimeError`** – When the computation is supported but fails.

`abstract property distributional`

Whether this is a distribution or distributional ambiguity set.

If this is `True`, then this represents a random distribution (stochastic programming) or an ambiguity set of random distributions (distributionally robust optimization) and any expression that depends on its random *`parameter`*, when used in a constraint or as an objective function, is understood as a (worst-case) *expected* value.

If this is `False`, then this represents a perturbation set (robust optimization) and any expression that depends on its perturbation *`parameter`*, when used in a constraint or as an objective function, is understood as a worst-case value.

`abstract property parameter`

The perturbation parameter.

`property subtype`

`property type`

Detailed type of a perturbation parameter universe.

`class picos.expressions.uncertain.perturbation.PerturbationUniverseType`(*theClass*, *subtype*)

Bases: *`DetailedType`*

Container for a pair of perturbation universe class type and subtype.

picos.expressions.uncertain.uexp_affine

Implements *`UncertainAffineExpression`*.

Classes

`class picos.expressions.uncertain.uexp_affine.UncertainAffineExpression`(*string*, *shape=(1, 1)*, *coefficients={}*)

Bases: *`UncertainExpression`*, *`BiaffineExpression`*

A multidimensional uncertain affine expression.

This expression has the form

$$A(x, \theta) = B(x, \theta) + P(x) + Q(\theta) + C$$

where B, P, Q, C and x are defined as for the *`BiaffineExpression`* base class and θ is an uncertain perturbation parameter confined to (distributed according to) a perturbation set (distribution) Θ.

If no coefficient matrices defining B and P are provided, then this expression represents uncertain data confined to an uncertainty set $\{Q(\theta) + C \mid \theta \in \Theta\}$ (distributed according to $Q(\Theta) + C$) where C can be understood as a nominal data value while $Q(\theta)$ quantifies the uncertainty on the data.

`__abs__`()

Denote the Euclidean or Frobenius norm of the expression.

`__ge__`(*other*)

Return self>=value.

`__init__`(*string*, *shape=(1, 1)*, *coefficients={}*)

Construct an `UncertainAffineExpression`.

Extends *`exp_biaffine.BiaffineExpression.__init__`*.

This constructor is meant for internal use. As a user, you will want to first define a universe (e.g. `ConicPerturbationSet`) for a `perturbation parameter` and use that parameter as a building block to create more complex uncertain expressions.

`__le__`(*other*)

Return self<=value.

picos.expressions.uncertain.uexp_norm

Implements `UncertainNorm`.

Classes

`class picos.expressions.uncertain.uexp_norm.`**`UncertainNorm`**(*x*)

Bases: `UncertainExpression`, `Expression`

Euclidean or Frobenius norm of an uncertain affine expression.

`__init__`(*x*)

Construct an `UncertainNorm`.

Parameters

x (`UncertainAffineExpression`) – The uncertain affine expression to denote the norm of.

`__le__`(*other*)

Return self<=value.

`__pow__`(*other*)

Denote exponentiation with another, scalar expression.

`property x`

Uncertain affine expression under the norm.

picos.expressions.uncertain.uexp_rand_pwl

Implements `RandomExtremumAffine`.

Classes

class picos.expressions.uncertain.uexp_rand_pwl.**RandomExtremumAffine**(*expressions*)

Bases: *ExtremumBase*, *UncertainExpression*, *Expression*

Base class for random convex or concave piecewise linear expressions.

Note: Unlike other uncertain expression types, this class is limited to uncertainty of stochastic nature, where using the expression in a constraint or as an objective function implicitly takes the (worst-case) expectation of the expression. Non-stochastic uncertainty is handled within *MaximumConvex* and *MinimumConcave* as their behavior, although designed for certain expression types, already encodes the worst-case approach of the robust optimization paradigm.

__ge__(*other*)

Return self>=value.

__init__(*expressions*)

Construct a *RandomExtremumAffine*.

Parameters

expressions – A collection of uncertain affine expressions whose uncertainty is of stochastic nature.

__le__(*other*)

Return self<=value.

property expressions

The expressions under the extremum.

property perturbation

Fast override for *UncertainExpression*.

class picos.expressions.uncertain.uexp_rand_pwl.**RandomMaximumAffine**(*expressions*)

Bases: *MaximumBase*, *RandomExtremumAffine*

The maximum over a set of random affine expressions.

class picos.expressions.uncertain.uexp_rand_pwl.**RandomMinimumAffine**(*expressions*)

Bases: *MinimumBase*, *RandomExtremumAffine*

The minimum over a set of random affine expressions.

picos.expressions.uncertain.uexp_sqnorm

Implements *UncertainSquaredNorm*.

Classes

class picos.expressions.uncertain.uexp_sqnorm.**UncertainSquaredNorm**(x)

Bases: *UncertainExpression*, *Expression*

Squared Euclidean or Frobenius norm of an uncertain affine expression.

__init__(x)

Construct an *UncertainSquaredNorm*.

Parameters

x (UncertainAffineExpression) – The uncertain affine expression to denote the squared norm of.

`__le__`(*other*)

Return self<=value.

`property x`

Uncertain affine expression under the squared norm.

picos.expressions.uncertain.uexpression

Implements the *`UncertainExpression`* base class.

Exceptions

`exception` `picos.expressions.uncertain.uexpression.`**`IntractableWorstCase`**

Bases: `RuntimeError`

Computing a worst-case (expected) value is hard and not supported.

Raised by *`worst_case`* and methods that depend on it.

Classes

`class` `picos.expressions.uncertain.uexpression.`**`UncertainExpression`**

Bases: `object`

Primary base class for uncertainty affected expression types.

The secondary base class must be *`Expression`* or a subclass thereof.

Uncertain expressions have a distinct behavior when used to form a constraint or when posed as an objective function. The exact behavior depends on the type of uncertainty involved. If the perturbation parameter that describes the uncertainty is confied to a perturbation set, then the worst-case realization of the parameter is assumed when determining feasibility and optimality. If the perturbation parameter is a random variable (whose distribution may itself be ambiguous), then the constraint or objective implicitly considers the expected value of the uncertain expression (under the worst-case distribution). Uncertain expressions are thus used in the contexts of robust optimization, stochastic programming and distributionally robust optimization.

`worst_case`(*direction*)

Find a worst-case realization of the uncertainty for the expression.

Expressions that are affected by uncertainty are only partially valued once an optimization solution has been applied. While their decision values are populated with a robust optimal solution, the parameter that controls the uncertainty is not valued unless the user assigned it a particular realization by hand. This method computes a worst-case (expected) value of the expression and returns it together with a realization of the perturbation parameter for which the worst case is attained (or `None` in the case of stochastic uncertainty).

For multidimensional expressions, this method computes the entrywise worst case and returns an attaining realization for each entry.

Parameters

`direction` (*`str`*) – Either `"min"` or `"max"`, denoting the worst-case direction.

Returns

A pair `(value, realization)`. For a scalar expression, `value` is its worst-case (expected) value as a `float` and `realization` is a realization of the *`perturbation`* parameter that attains this worst case as a `float` or CVXOPT matrix. For a multidimensional expression, `value` is a CVXOPT dense matrix denoting the entrywise worst-case values and `realization` is a `tuple` of attaining realizations corresponding to the expression vectorized in in column-major order. Lastly, `realization` is `None` if the expression is *`certain`* or when its uncertainty is of stochastic nature.

Raises

- *`picos.NotValued`* – When the decision variables that occur in the expression are not fully valued.
- *`picos.uncertain.IntractableWorstCase`* – When computing the worst-case (expected) value is not supported, in particular when it would require solving a nonconvex problem.
- `RuntimeError` – When the computation is supported but fails.

`worst_case_string`(*direction*)

A string describing the expression within a worst-case context.

Parameters
direction (*`str`*) – Either "`min`" or "`max`", denoting the worst-case direction.

`worst_case_value`(*direction*)

A shorthand for the first value returned by *`worst_case`*.

`property certain`

Whether the uncertain expression is actually certain.

`property perturbation`

The parameter controlling the uncertainty, or `None`.

`property random`

Whether the uncertainty is of stochastic nature.

See also *`distributional`*.

`property uncertain`

Whether the uncertain expression is in fact uncertain.

`property universe`

Universe that the perturbation parameter lives in, or `None`.

If this is not `None`, then this is the same as *`perturbation`*.:attr:~*.perturbation.Perturbation.universe*.

picos.expressions.variables

Implements all mathematical variable types and their base class.

Classes

`class picos.expressions.variables.BaseVariable`(*name*, *vectorization*, *lower=None*, *upper=None*)

Bases: *`Mutable`*

Primary base class for all variable types.

Variables need to inherit this class with priority (first class listed) and *`ComplexAffineExpression`* or *`AffineExpression`* without priority.

`__init__`(*name*, *vectorization*, *lower=None*, *upper=None*)

Perform basic initialization for *`BaseVariable`* instances.

Parameters

- **name** (*`str`*) – Name of the variable. A leading "__" denotes a private variable and is replaced by a sequence containing the variable's unique ID.

- **vectorization** (BaseVectorization) – Vectorization format used to store the value.
- **lower** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **upper** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

copy(*new_name=None*)

Return an independent copy of the variable.

classmethod make_var_type(**args*, ***kwargs*)

Create a detailed variable type from subtype parameters.

See also *var_type*.

property bound_constraint

The variable bounds as a PICOS constraint, or `None`.

property bound_dicts

Variable bounds as a pair of mappings from index to scalar bound.

The indices and bound values are with respect to the internal representation of the variable, whose value can be accessed with *internal_value*.

Upper and lower bounds set to `float("inf")` and `float("-inf")` on variable creation, respectively, are not included.

property long_string

Long string representation for printing a *Problem*.

property num_bounds

Number of scalar bounds associated with the variable.

property var_subtype

The subtype part of the detailed variable type.

See also *var_type*.

property var_type

The detailed variable type.

This intentionally does not override *Expression.type* so that the variable still behaves as the affine expression that it represents when prediction constraint outcomes.

class picos.expressions.variables.**BinaryVariable**(*name*, *shape=(1, 1)*)

Bases: *BaseVariable*, *AffineExpression*

A $\{0, 1\}$-valued variable.

__init__(*name*, *shape=(1, 1)*)

Create a *BinaryVariable*.

Parameters

- **name** (*str*) – The variable's name, used for both string description and identification.
- **shape** (*int or tuple or list*) – The shape of a vector or matrix variable.

class picos.expressions.variables.**ComplexVariable**(*name*, *shape=(1, 1)*)

Bases: *BaseVariable*, *ComplexAffineExpression*

A complex-valued variable.

Passed to solvers as a real variable vector with $2mn$ entries.

`__init__`(*name*, *shape=(1, 1)*)

Create a `ComplexVariable`.

Parameters

- **`name`** (`str`) – The variable's name, used for both string description and identification.
- **`shape`** (`int or tuple or list`) – The shape of a vector or matrix variable.

`class picos.expressions.variables.`**`HermitianVariable`**(*name*, *shape*)

Bases: `BaseVariable`, `ComplexAffineExpression`

A hermitian matrix variable.

Stored internally and passed to solvers as the horizontal concatenation of a real symmetric vectorization with $\frac{n(n+1)}{2}$ entries and a real skew-symmetric vectorization with $\frac{n(n-1)}{2}$ entries, resulting in a real vector with only n^2 entries total.

`__init__`(*name*, *shape*)

Create a `HermitianVariable`.

Parameters

- **`name`** (`str`) – The variable's name, used for both string description and identification.
- **`shape`** (`int or tuple or list`) – The shape of the matrix.

`class picos.expressions.variables.`**`IntegerVariable`**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: `BaseVariable`, `AffineExpression`

An integer-valued variable.

`__init__`(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create an `IntegerVariable`.

Parameters

- **`name`** (`str`) – The variable's name, used for both string description and identification.
- **`shape`** (`int or tuple or list`) – The shape of a vector or matrix variable.
- **`lower`** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **`upper`** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

`class picos.expressions.variables.`**`LowerTriangularVariable`**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: `BaseVariable`, `AffineExpression`

A lower triangular matrix variable.

Stored internally and passed to solvers as a lower triangular vectorization with only $\frac{n(n+1)}{2}$ entries.

`__init__`(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create a `LowerTriangularVariable`.

Parameters

- **`name`** (`str`) – The variable's name, used for both string description and identification.
- **`shape`** (`int or tuple or list`) – The shape of the matrix.
- **`lower`** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **`upper`** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

class picos.expressions.variables.**RealVariable**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: *BaseVariable*, *AffineExpression*

A real-valued variable.

__init__(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create a *RealVariable*.

Parameters

- **name** (*str*) – The variable's name, used for both string description and identification.
- **shape** (*int or tuple or list*) – The shape of a vector or matrix variable.
- **lower** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **upper** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

class picos.expressions.variables.**SkewSymmetricVariable**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: *BaseVariable*, *AffineExpression*

A skew-symmetric matrix variable.

Stored internally and passed to solvers as a skew-symmetric vectorization with only $\frac{n(n-1)}{2}$ entries.

__init__(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create a *SkewSymmetricVariable*.

Parameters

- **name** (*str*) – The variable's name, used for both string description and identification.
- **shape** (*int or tuple or list*) – The shape of the matrix.
- **lower** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **upper** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

class picos.expressions.variables.**SymmetricVariable**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: *BaseVariable*, *AffineExpression*

A symmetric matrix variable.

Stored internally and passed to solvers as a symmetric vectorization with only $\frac{n(n+1)}{2}$ entries.

__init__(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create a *SymmetricVariable*.

Parameters

- **name** (*str*) – The variable's name, used for both string description and identification.
- **shape** (*int or tuple or list*) – The shape of the matrix.
- **lower** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **upper** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

class picos.expressions.variables.**UpperTriangularVariable**(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Bases: *BaseVariable*, *AffineExpression*

An upper triangular matrix variable.

Stored internally and passed to solvers as an upper triangular vectorization with only $\frac{n(n+1)}{2}$ entries.

__init__(*name*, *shape=(1, 1)*, *lower=None*, *upper=None*)

Create a *UpperTriangularVariable*.

Parameters

- **name** (*str*) – The variable's name, used for both string description and identification.
- **shape** (*int or tuple or list*) – The shape of the matrix.
- **lower** – Constant lower bound on the variable. May contain `float("-inf")` to denote unbounded elements.
- **upper** – Constant upper bound on the variable. May contain `float("inf")` to denote unbounded elements.

class picos.expressions.variables.**VariableType**(*theClass*, *subtype*)

Bases: *DetailedType*

The detailed type of a variable for predicting reformulation outcomes.

Objects

picos.expressions.variables.**CONTINUOUS_VARTYPES**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

Default value

```
(<class 'picos.expressions.variables.RealVariable'>,
 <class 'picos.expressions.variables.ComplexVariable'>,
 <class 'picos.expressions.variables.SymmetricVariable'>,
 <class 'picos.expressions.variables.SkewSymmetricVariable'>,
 <class...
```

picos.expressions.vectorizations

Implement special matrix vectorization formats.

These formats are used to efficiently store structured mutable types such as symmetric matrix variables in the form of real vectors.

Classes

class picos.expressions.vectorizations.**BaseVectorization**(*shape*)

Bases: `ABC`, `object`

Abstract base class for special matrix vectorization formats.

Subclass instances are cached: If multiple instances of the same vectorization format and concerning matrices of the same shape are requested successively, then the instance created to serve the first request is retrieved from a cache on successive requests. The module attributes *`CACHE_SIZE`* and *`CACHE_BULK_REMOVE`* control the size of the cache for each vectorization format.

Warning: Due to how caching is implemented, derived classes may not inherit from each other but only from *`BaseVectorization`* directly!

__init__(*shape*)

Initialize a vectorization format for a fixed matrix shape.

static **__new__**(*cls*, *shape*)

Lookup or create a vectorization format for a fixed matrix shape.

devectorize(*vector*)

Given a special vectorization, return the corresponding matrix.

Raises

`TypeError` – If the input isn't a CVXOPT column vector or does not have the expected numeric type or length.

vectorize(*matrix*)

Given a matrix, return its special vectorization.

Raises

- **`TypeError`** – If the input isn't a CVXOPT matrix or does not have the expected numeric type or shape.
- **`ValueError`** – If the matrix does not have the expected structure.

property **dim**

The length of the vectorization.

This corresponds to the dimension of a matrix mutable being vectorized.

property **identity**

A linear mapping from the special to the full vectorization.

The term *identity* comes from the fact that these matrices are used as the coefficients that map the internal (vectorized) representation of a *`Mutable`* object to the *`BiaffineExpression`* instance that represents the mutable in algebraic operations.

property **shape**

The shape of matrices being vectorized.

class picos.expressions.vectorizations.**ComplexVectorization**(*shape*)

Bases: *`BaseVectorization`*

An isometric vectorization that stacks real and imaginary parts.

vectorize(*matrix*)

Override *`BaseVectorization.vectorize`*.

This is necessary because extracting the real and the imaginary part cannot be done with a linear transformation matrix as is done for other vectorization formats.

class picos.expressions.vectorizations.**FullVectorization**(*shape*)

Bases: *BaseVectorization*

A basic column-major matrix vectorization.

devectorize(*vector*)

Override *BaseVectorization.devectorize* for speed reasons.

vectorize(*matrix*)

Override *BaseVectorization.vectorize* for speed reasons.

class picos.expressions.vectorizations.**HermitianVectorization**(*shape*)

Bases: *BaseVectorization*

An isometric hermitian matrix vectorization.

The vectorization is isometric with respect to the Hermitian inner product $\langle A, B \rangle = \operatorname{tr}(B^H A)$ on the matrices and the real dot product on their vectorizations.

__init__(*shape*)

Initialize a vectorization format for hermitian matrices.

Uses *SymmetricVectorization* (for the real part) and *SkewSymmetricVectorization* (for the imaginary part) internally.

vectorize(*matrix*)

Override *BaseVectorization.vectorize*.

This is necessary because extracting the real and the imaginary part cannot be done with a linear transformation matrix as is done for other vectorization formats.

class picos.expressions.vectorizations.**LowerTriangularVectorization**(*shape*)

Bases: *BaseVectorization*

An isometric lower triangular matrix vectorization.

class picos.expressions.vectorizations.**SkewSymmetricVectorization**(*shape*)

Bases: *BaseVectorization*

An isometric skew-symmetric matrix vectorization.

class picos.expressions.vectorizations.**SymmetricVectorization**(*shape*)

Bases: *BaseVectorization*

An isometric symmetric matrix vectorization.

See [svec] for the precise vectorization used.

class picos.expressions.vectorizations.**UpperTriangularVectorization**(*shape*)

Bases: *BaseVectorization*

An isometric upper triangular matrix vectorization.

Objects

picos.expressions.vectorizations.**CACHE_BULK_REMOVE**

Number of cached instances to drop at random when the cache is full.

Default value

```
25
```

`picos.expressions.vectorizations.CACHE_SIZE`

Number of instances to cache per vectorization format.

Default value

```
100
```

6.1.6 picos.formatting

Console output helpers related to formatting.

Functions

`picos.formatting.arguments`(*strings*, *sep=', '*, *empty=''*)

A wrapper around *parameterized_string* for argument lists.

Parameters

- **strings** (*list(str)*) – String descriptions of the arguments.
- **sep** (*str*) – Separator.
- **empty** (*str*) – Representation of an empty argument list.

`picos.formatting.box`(*title*, *caption*, *subtitle=None*, *symbol='-'*, *width=HEADER_WIDTH*, *show=True*)

Print both a header above and a footer below the context.

Parameters

- **title** (*str*) – The (long) main title printed at the top.
- **caption** (*str*) – The (short) caption printed at the bottom.
- **subtitle** (*str*) – The (long) subtitle printed at the top.
- **symbol** (*str*) – A single character used to draw lines.
- **width** (*int*) – The width of the box.
- **show** (*bool*) – Whether anything should be printed at all.

`picos.formatting.detect_range`(*sequence*, *asQuadruple=False*, *asStringTemplate=False*, *shortString=False*)

Return a Python range mirroring the given integer 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.formatting 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)
'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.formatting.`**`doc_cat`**(*docstring*, *append*)

Append to a docstring.

`picos.formatting.`**`natsorted`**(*strings*, *key=None*)

Sort the given list of strings naturally with respect to numbers.

`picos.formatting.`**`parameterized_string`**(*strings*, *replace='-?\\d+'*, *context='\\W'*, *placeholders='ijklpqr'*, *fallback='?'*)

Find a string template for the given (algebraic) strings.

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.
- **context** (*str*) – A regular expression describing context characters that need to be present next to the bits to be replaced 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.formatting import parameterized_string as ps
>>> ps(["A[{}]".format(i) for i in range(5, 31)])
('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.formatting.**picos_box**(*show=True*)

Print a PICOS header above and a PICOS footer below the context.

picos.formatting.**print_footer**(*caption*, *symbol='='*, *width=HEADER_WIDTH*)

Print a footer line.

picos.formatting.**print_header**(*title*, *subtitle=None*, *symbol='-'*, *width=HEADER_WIDTH*)

Print a header line.

picos.formatting.**solver_box**(*longName*, *shortName*, *subSolver=None*, *show=True*)

Print a solver header above and a solver footer below the context.

Objects

picos.formatting.**HEADER_WIDTH**

Character length of headers and footers printed by PICOS.

Default value

```
35
```

6.1.7 picos.glyphs

String templates used to print (algebraic) expressions.

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.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()
```

Classes

class picos.glyphs.Am(*glyph*)

Bases: *Op*

A math atom glyph.

__init__(*glyph*)

Construct an *Am* glyph.

Parameters
glyph (*str*) – The glyph's format string template.

class picos.glyphs.Br(*glyph*)

Bases: *Op*

A math operator glyph with enclosing brackets.

__init__(*glyph*)

Construct a *Br* glyph.

Parameters
glyph (*str*) – The glyph's format string template.

class picos.glyphs.Fn(*glyph*)

Bases: *Op*

A math operator glyph in function form.

__init__(*glyph*)

Construct a *Fn* glyph.

Parameters
glyph (*str*) – The glyph's format string template.

class picos.glyphs.Gl(*glyph*)

Bases: `object`

The basic "glyph", an (algebraic) string formatting template.

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.

__init__(*glyph*)

Construct a glyph.

Parameters
glyph (*str*) – The glyph's format string template.

rebuild()

If the template was created using other glyphs, rebuild it.

Returns
True if the template has changed.

reset()

Reset the glyph to its initial formatting template.

update(*new*)

Change the glyph's formatting template.

`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.

`__init__`(*string*, *glyph*, *operands*)

Augment the Python string with metadata on its origin.

`static` **`__new__`**(*cls*, *string*, *glyph*, *operands*)

Create a regular Python string.

`reglyphed`(*replace={}*)

Returns a rebuilt version of the string using current glyphs.

Parameters

`replace` (*`dict`*) – Replace leaf-node (non *`GlStr`*) strings with new strings. This can be used, for instance, to change the names of varaibles.

`glyph`

The glyph used to create the string.

`operands`

The operands used to create the string.

`class picos.glyphs.Op`(*glyph*, *order*, *assoc=False*, *closed=False*)

Bases: *`Gl`*

The basic math operator glyph.

`__init__`(*glyph*, *order*, *assoc=False*, *closed=False*)

Construct a math operator glyph.

Parameters

- **`glyph`** (*`str`*) – The glyph's format string template.
- **`order`** (*`int`*) – The operator's position in the binding strength hierarchy. Operators with lower numbersbind 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 or list(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 specified per operand in the form of a list.

`reset`()

Reset the glyph to its initial behavior.

`update`(*new*)

Change the glyph's behavior.

`class picos.glyphs.OpStr`(*string*, *glyph*, *operands*)

Bases: *`GlStr`*

A string created from a math operator glyph.

`class picos.glyphs.Rl(glyph)`

Bases: `Op`

A math relation glyph.

`__init__(glyph)`

Construct a `Rl` glyph.

Parameters

glyph (`str`) – The glyph's format string template.

`class picos.glyphs.Tr(glyph)`

Bases: `Op`

A math glyph in superscript/trailer form.

`__init__(glyph)`

Construct a `Tr` glyph.

Parameters

glyph (`str`) – The glyph's format string template.

Functions

`picos.glyphs.ascii()`

Let PICOS create future strings using only ASCII characters.

`picos.glyphs.clever_add(left, right)`

Describe the addition of two values in a clever way.

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, clever_add, matrix
>>> add("x", neg("y"))
'x + -y'
>>> clever_add("x", neg("y"))
'x - y'
>>> add("X", matrix(neg("y")))
'X + [-y]'
>>> clever_add("X", matrix(neg("y")))
'X - [y]'
>>> clever_add("X", matrix(-1.5))
'X - [1.5]'
```

`picos.glyphs.clever_div(left, right)`

Describe the division of one value by another in a clever way.

A wrapper around `div` that factors out identity factors.

`picos.glyphs.clever_dotp(left, right, complexLHS, scalar=False)`

Describe an inner product in a clever way.

Parameters

complexRHS (`bool`) – Whether the right hand side is complex.

`picos.glyphs.clever_mul(left, right)`

Describe the multiplocation of two values in a clever way.

A wrapper around `mul` that factors out identity factors.

`picos.glyphs.clever_neg(value)`

Describe the negation of a value in a clever way.

A wrapper around `neg` that resorts to unnegating an already negated value.

Example

```
>>> from picos.glyphs import neg, clever_neg, matrix
>>> neg("x")
'-x'
>>> neg(neg("x"))
'-(-x)'
>>> clever_neg(neg("x"))
'x'
>>> neg(matrix(-1))
'-[-1]'
>>> clever_neg(matrix(-1))
'[1]'
```

`picos.glyphs.clever_sub(left, right)`

Describe the substraction of a value from another in a clever way.

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, clever_sub, matrix
>>> sub("x", neg("y"))
'x - -y'
>>> clever_sub("x", neg("y"))
'x + y'
>>> sub("X", matrix(neg("y")))
'X - [-y]'
>>> clever_sub("X", matrix(neg("y")))
'X + [y]'
>>> clever_sub("X", matrix(-1.5))
'X + [1.5]'
```

`picos.glyphs.col_vectorize(*entries)`

Describe a column vector with the given symbolic entries.

`picos.glyphs.default(rebuildDerivedGlyphs=True)`

Let PICOS create future strings using only unicode characters.

`picos.glyphs.free_var_name(string)`

Return a variable name not present in the given string.

`picos.glyphs.from_glyph(string, theGlyph)`

Whether the given string was created by the given glyph.

`picos.glyphs.is_negated(value)`

Check if a value can be unnegated by `unnegate`.

`picos.glyphs.latin1(rebuildDerivedGlyphs=True)`

Let PICOS create future strings using only ISO 8859-1 characters.

`picos.glyphs.make_function(*names)`

Create an ad-hoc composite function glyphs.

Example

```
>>> from picos.glyphs import make_function
>>> make_function("log", "sum", "exp")("x")
'log∘sum∘exp(x)'
```

picos.glyphs.**matrix_cat**(*left*, *right*, *horizontal=True*)

Describe matrix concatenation in a clever way.

A wrapper around *matrix*, *horicat* and *vertcat*.

Example

```
>>> from picos.glyphs import matrix_cat
>>> Z = matrix_cat("X", "Y")
>>> Z
'[X, Y]'
>>> matrix_cat(Z, Z)
'[X, Y, X, Y]'
>>> matrix_cat(Z, Z, horizontal = False)
'[X, Y; X, Y]'
```

picos.glyphs.**rebuild**()

Update glyphs that are based upon other glyphs.

picos.glyphs.**row_vectorize**(**entries*)

Describe a row vector with the given symbolic entries.

picos.glyphs.**scalar**(*value*)

Format a scalar value.

This function mimics an operator glyph, but it returns a normal string (as opposed to an *OpStr*) for nonnegative numbers.

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.**shape**(*theShape*)

Describe a matrix shape that can contain wildcards.

A wrapper around *size* that takes just one argument (the shape) that may contain wildcards which are printed as `'?'`.

picos.glyphs.**show**(**args*)

Show output from all glyphs.

Parameters

args (*list(str)*) – Strings to use as glyph operands.

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

Let PICOS create future strings using only unicode characters.

picos.glyphs.**unnegate**(*value*)

Unnegate a value, usually a glyph-created string, in a sensible way.

Unnegates a *operator glyph created string* or other 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* returns `False`.

Objects

picos.glyphs.**CAN_FACTOR_OUT_NEGATION**

Operator glyphs for which negation may be factored out.

Default value

```
(<picos.glyphs.Br object at 0x7d3777562d50>,
 <picos.glyphs.Fn object at 0x7d3777563020>,
 <picos.glyphs.Fn object at 0x7d3777563230>,
 <picos.glyphs.Fn object at 0x7d3777563140>,
 <picos.glyphs.Fn object at 0x7d3777563260>,
 <picos.glyphs.Fn object...
```

picos.glyphs.**abs**

Absolute value glyph.

Default value

```
<picos.glyphs.Br object at 0x7d3777562db0>
```

picos.glyphs.**add**

Addition glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563770>
```

picos.glyphs.**affpart**

Affine part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775633b0>
```

picos.glyphs.**and_**

Logical and glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562c60>
```

picos.glyphs.**bcasted**

Broadcasted glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563530>
```

picos.glyphs.**blinpart**

Bilinear part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563410>
```

picos.glyphs.**closure**

Set closure glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777562ab0>
```

picos.glyphs.**comma**

Seperated by comma glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d37775629f0>
```

picos.glyphs.**compose**

Function composition glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562a50>
```

picos.glyphs.**compsep**

Compact seperator glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d37775629c0>
```

picos.glyphs.**conj**

Complex conugate glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563350>
```

picos.glyphs.**cstpart**

Constant part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563470>
```

picos.glyphs.**cubed**

Cubed value glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d37775639b0>
```

picos.glyphs.**desvec**

Symmetric de-vectorization glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563200>
```

picos.glyphs.**det**

Determinant glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775632c0>
```

`picos.glyphs.diag`

Diagonal matrix glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563260>
```

`picos.glyphs.div`

Division glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563860>
```

`picos.glyphs.dotp`

Scalar product glyph.

Default value

```
<picos.glyphs.Br object at 0x7d3777562d80>
```

`picos.glyphs.element`

Set element glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563b90>
```

`picos.glyphs.eq`

Equality glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563bc0>
```

`picos.glyphs.exp`

Exponentiation glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775630e0>
```

`picos.glyphs.exparg`

Expected value glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563740>
```

`picos.glyphs.expected`

Epected value glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775635f0>
```

`picos.glyphs.forall`

Universal quantification glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562c00>
```

picos.glyphs.**fromto**

Range glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562b10>
```

picos.glyphs.**frozen**

Frozen mutables glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775634a0>
```

picos.glyphs.**ge**

Greater or equal glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563bf0>
```

picos.glyphs.**geomean**

Matrix geometric mean glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775638f0>
```

picos.glyphs.**gt**

Greater than glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563c20>
```

picos.glyphs.**hadamard**

Hadamard product glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775637d0>
```

picos.glyphs.**horicat**

Horizontal concatenation glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563b30>
```

picos.glyphs.**htransp**

Matrix hermitian transposition glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563ad0>
```

picos.glyphs.**idmatrix**

Identity matrix glyph.

Default value

```
<picos.glyphs.Am object at 0x7d3777562cf0>
```

`picos.glyphs.imag`

Imaginary part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563320>
```

`picos.glyphs.index`

Index glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563b00>
```

`picos.glyphs.infty`

Infinity glyph.

Default value

```
<picos.glyphs.Am object at 0x7d3777562cc0>
```

`picos.glyphs.interval`

Interval glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562ae0>
```

`picos.glyphs.intrange`

Integer range glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562b70>
```

`picos.glyphs.inverse`

Matrix inverse glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563a10>
```

`picos.glyphs.kron`

Kronecker product glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563800>
```

`picos.glyphs.lambda_`

Lambda symbol glyph.

Default value

```
<picos.glyphs.Am object at 0x7d3777562d20>
```

`picos.glyphs.le`

Lesser or equal glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563c50>
```

picos.glyphs.**leadsto**

Successorship glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562c30>
```

picos.glyphs.**linpart**

Linear part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775633e0>
```

picos.glyphs.**log**

Logarithm glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563110>
```

picos.glyphs.**lt**

Lesser than glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563c80>
```

picos.glyphs.**maindiag**

Main diagonal glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563290>
```

picos.glyphs.**matrix**

Matrix glyph.

Default value

```
<picos.glyphs.Br object at 0x7d3777562d50>
```

picos.glyphs.**max**

Maximum glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563080>
```

picos.glyphs.**maxarg**

The basic math operator glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563d10>
```

picos.glyphs.**min**

Minimum glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775630b0>
```

picos.glyphs.**minarg**

The basic math operator glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563d40>
```

picos.glyphs.**mul**

Multiplication glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563830>
```

picos.glyphs.**ncnorm**

Nuclear Norm glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777562fc0>
```

picos.glyphs.**ncstpart**

Nonconstant part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563440>
```

picos.glyphs.**neg**

Negation glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563890>
```

picos.glyphs.**norm**

Norm glyph.

Default value

```
<picos.glyphs.Br object at 0x7d3777562de0>
```

picos.glyphs.**or_**

Logical or glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562c90>
```

picos.glyphs.**parenth**

Parenthesis glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562960>
```

picos.glyphs.**plsmns**

Plus/Minus glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775638c0>
```

picos.glyphs.**pnorm**

p-Norm glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777562e40>
```

picos.glyphs.**power**

Power glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563950>
```

picos.glyphs.**pqnorm**

pq-Norm glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777562ea0>
```

picos.glyphs.**probdist**

Probability distribution glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775635c0>
```

picos.glyphs.**prod**

Product glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563050>
```

picos.glyphs.**psdge**

Lesser or equal w.r.t. the p.s.d. cone glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563cb0>
```

picos.glyphs.**psdle**

Greater or equal w.r.t. the p.s.d. cone glyph.

Default value

```
<picos.glyphs.Rl object at 0x7d3777563ce0>
```

picos.glyphs.**ptrace**

Matrix p-Trace glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563680>
```

picos.glyphs.**ptrace_**

Matrix partial trace glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563710>
```

picos.glyphs.**ptransp**

Matrix partial transposition glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563aa0>
```

picos.glyphs.**ptransp_**

Matrix partial transposition glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775636e0>
```

picos.glyphs.**qe**

Quantum entropy glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563620>
```

picos.glyphs.**qre**

Quantum relative entropy glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563650>
```

picos.glyphs.**quadpart**

Quadratic part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563380>
```

picos.glyphs.**real**

Real part glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775632f0>
```

picos.glyphs.**repr1**

Representation glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d37775628d0>
```

picos.glyphs.**repr2**

Long representation glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562930>
```

picos.glyphs.**reshaped**

Column-major (Fortran) reshaped glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775634d0>
```

`picos.glyphs.reshaprm`

Row-major (C-order) reshaped glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563500>
```

`picos.glyphs.sep`

Seperator glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562990>
```

`picos.glyphs.set`

Set glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562a80>
```

`picos.glyphs.shortint`

Shortened interval glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562bd0>
```

`picos.glyphs.shuffled`

Matrix reshuffling glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563590>
```

`picos.glyphs.size`

Matrix size/shape glyph.

Default value

```
<picos.glyphs.Gl object at 0x7d3777562a20>
```

`picos.glyphs.slice`

Expression slicing glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775636b0>
```

`picos.glyphs.spnorm`

Spectral Norm glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777562f30>
```

`picos.glyphs.sqrt`

Square root glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563560>
```

`picos.glyphs.squared`

Squared value glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d37775639e0>
```

`picos.glyphs.sub`

Substraction glyph.

Default value

```
<picos.glyphs.Op object at 0x7d37775637a0>
```

`picos.glyphs.sum`

Summation glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563020>
```

`picos.glyphs.svec`

Symmetric vectorization glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775631d0>
```

`picos.glyphs.trace`

Matrix trace glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563230>
```

`picos.glyphs.transp`

Matrix transposition glyph.

Default value

```
<picos.glyphs.Tr object at 0x7d3777563a40>
```

`picos.glyphs.trilvec`

Lower triangular vectorization glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563170>
```

`picos.glyphs.triuvec`

Upper triangular vectorization glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d37775631a0>
```

`picos.glyphs.vec`

Vectorization glyph.

Default value

```
<picos.glyphs.Fn object at 0x7d3777563140>
```

`picos.glyphs.`**`vertcat`**

Vertical concatenation glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563b60>
```

`picos.glyphs.`**`wgeomean`**

Weighted matrix geometric mean glyph.

Default value

```
<picos.glyphs.Op object at 0x7d3777563920>
```

6.1.8 picos.legacy

Backwards-compatibility and deprecation helpers.

Functions

`picos.legacy.`**`deprecated`**`(`*since*, *reason=None*, *useInstead=None*, *decoratorLevel=0*`)`

Decorate a deprecated function or method.

Parameters

- **`reason`** (*`str`*) – Textual deprecation reason.
- **`useInstead`** (*`str`*) – Alternative to the deprecated object.
- **`decoratorLevel`** (*`int`*) – If not the only decorator, number of decorators above this one that add to the stack level of a warning thrown.

`picos.legacy.`**`map_legacy_options`**`(`*options={}*, ***kwOptions*`)`

Map deprecated solver options to those replacing them.

`picos.legacy.`**`throw_deprecation_warning`**`(`*reason*, *decoratorLevel=0*`)`

Raise a deprecation warning.

6.1.9 picos.modeling

Optimization problem modeling toolbox.

Exceptions

`exception` `picos.modeling.`**`NoStrategyFound`**

See *`picos.modeling.strategy.NoStrategyFound`*.

`exception` `picos.modeling.`**`SolutionFailure`**

See *`picos.modeling.problem.SolutionFailure`*.

Classes

class picos.modeling.**Footprint**

See *picos.modeling.footprint.Footprint*.

class picos.modeling.**Objective**

See *picos.modeling.objective.Objective*.

class picos.modeling.**Options**

See *picos.modeling.options.Options*.

class picos.modeling.**Problem**

See *picos.modeling.problem.Problem*.

class picos.modeling.**Solution**

See *picos.modeling.solution.Solution*.

class picos.modeling.**Specification**

See *picos.modeling.footprint.Specification*.

class picos.modeling.**Strategy**

See *picos.modeling.strategy.Strategy*.

Functions

picos.modeling.**find_assignment**()

See *picos.modeling.quicksolve.find_assignment*.

picos.modeling.**maximize**()

See *picos.modeling.quicksolve.maximize*.

picos.modeling.**minimize**()

See *picos.modeling.quicksolve.minimize*.

picos.modeling.file_in

Functions for reading optimization problems from a file.

Functions

picos.modeling.file_in.**import_cbf**(*filename*)

Create a *Problem* from a CBF file.

The created problem contains one (multidimensional) 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`.

Returns

A tuple `(P, x, X, params)` where

- P is the imported picos *Problem* object,
- `x` is a list of multidimensional variables representing the scalar variables found in the file,
- `X` is a list of symmetric variables representing the positive semidefinite variables found in the file, and
- `params` is a dictionary containing PICOS cosntants used to define the problem. Indexing is with respect to the blocks of variables as defined in the sections `VAR` and `CON` of the file.

picos.modeling.file_out

Functions for writing optimization problems to a file.

Functions

`picos.modeling.file_out.`**`devectorize`**(*vec*)

Create a matrix from a symmetric vectorization.

`picos.modeling.file_out.`**`write`**(*picos_problem*, *filename*, *writer='picos'*)

Write an optimization problem to a file.

Parameters

- **`P`** (`Problem`) – The problem to write.
- **`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.

Warning: This function is severly outdated and may fail or not function as advertised.

Objects

`picos.modeling.file_out.`**`INFINITY`**

A number deemed too large to appear in practice.

Default value

```
1e+16
```

picos.modeling.footprint

Optimization problem description classes.

This module implements a footprint of a single problem, storing number and types of expressions and constraints, and the specification of a problem class.

Classes

`class` `picos.modeling.footprint.`**`Footprint`**`(`*recordsOrDict*`)`

Bases: `RecordTree`

Statistics on an optimization problem.

`__init__``(`*recordsOrDict*`)`

Construct a `Footprint` from raw data.

See `picos.containers.RecordTree.__init__`. The `addValues` argument is fixed; only variable and constraint paths are added.

`classmethod from_problem``(`*problem*`)`

Create a footprint from a problem instance.

`classmethod from_types``(`*obj_dir*, *obj_func*, *vars=[]*, *cons=[]*, *nd_opts={}*`)`

Create a footprint from collections of detailed types.

Parameters

- **`obj_dir`** (`str`) – Objective direction.
- **`obj_func`** – Detailed objective function type.
- **`nd_opts`** (`list(str)`) – A dictionary mapping option names to nondefault values.

Parm list(tuple) vars

A list of pairs of detailed variable type and occurence count.

Parm list(tuple) cons

A list of pairs of detailed constraint type and occurence count.

`updated``(`*recordsOrDict*`)`

Override `updated`.

This method, just like `__init__`, does not take the additional `addValues` argument.

`with_extra_options``(`***extra_options*`)`

Return a copy with additional solution search options applied.

property constraints

A dictionary mapping detailed constraint types to their quantity.

property continuous

Whether no integral variable type is present.

property cost

A very rough measure on how expensive solving such a problem is.

This is logarithmic in the estimated size of the constraint matrix.

property direction

Objective function optimization direction.

property integer

Whether an integral variable type is present.

property nonconvex_quadratic_objective

Whether the problem has a nonconvex quadratic objective.

property nondefault_options

A dictionary mapping option names to their nondefault values.

> **Warning:** This property is cached for performance reasons, do not modify any mutable option value (make a deep copy instead)!

property objective

Detailed type of the objective function.

property options

An `Options` object.

> **Warning:** This property is cached for performance reasons, do not modify the returned object (make a `copy` instead)!

property size

Return the estimated size of the (dense) scalar constraint matrix.

property variables

A dictionary mapping detailed variable types to their quantity.

class picos.modeling.footprint.**Specification**(*objectives=None*, *variables=None*, *constraints=None*, *nondefault_options=None*)

Bases: `object`

Representation of a mathematical class of optimization problems.

__init__(*objectives=None*, *variables=None*, *constraints=None*, *nondefault_options=None*)

Create a specification from the given features.

The token `None` means no requirement.

mismatch_reason(*footprint*)

Give one string reason why the given footprint does not match.

picos.modeling.objective

Implementation of *Objective*.

Classes

class picos.modeling.objective.**Objective**(*direction=None*, *function=None*)

Bases: *Valuable*

An optimization objective composed of search direction and function.

Example

```
>>> from picos import Objective, RealVariable
>>> x = RealVariable("x")
>>> obj = Objective("min", x); obj
<Objective: minimize x>
>>> obj + x**2  # Add a term to the objective function.
<Objective: minimize x + x²>
>>> obj/2 + 2*obj  # Scale and combine two objectives.
<Objective: minimize x/2 + 2·x>
>>> -obj  # Flip the optimization direction.
<Objective: maximize -x>
```

__add__(*other*)

Denote the sum of two compatible objectives.

__eq__(*other*)

Report whether two objectives are the same.

__init__(*direction=None*, *function=None*)

Construct an optimization objective.

Parameters

- **direction** (*str*) – Case insensitive search direction string. One of
  - "min" or "minimize",
  - "max" or "maximize",
  - "find" or None (for a feasibility problem).
- **function** (Expression) – The objective function. Must be None for a feasibility problem.

__mul__(*other*)

Denote the product of the objective with an expression.

__neg__()

Return the negated objective with the search direction flipped.

__pos__()

Return the objective as-is.

__rmul__(*other*)

Denote the product of the objective with an expression.

__sub__(*other*)

Denote the difference of two compatible objectives.

__truediv__(*other*)

Denote division of the objective by an expression.

FIND = 'find'

Short string denoting a feasibility problem.

MAX = 'max'

Short string denoting a maximization problem.

MIN = 'min'

Short string denoting a minimization problem.

property direction

Search direction as a short string.

property feasibility

Whether the objective is "find an assignment".

property function

Objective function.

property normalized

The objective but with feasiblity posed as "minimize 0".

```
>>> from picos import Objective
>>> obj = Objective(); obj
<Objective: find an assignment>
>>> obj.normalized
<Objective: minimize 0>
```

property pair

Search direction and objective function as a pair.

picos.modeling.options

Optimization solver parameter handling.

Classes

class picos.modeling.options.Option(**args*, ***kwargs*)

Bases: `object`

Optimization solver option.

A single option that affects how a *Problem* is solved.

An initial instance of this class is built from each entry of the *OPTIONS* table to obtain the *OPTION_OBJS* tuple.

__init__(*name*, *argType*, *default*, *description*, *check=lambda x: ...*)

Initialize an *Option*.

See *OPTIONS*.

static __new__(*cls*, **args*, ***kwargs*)

Create a blank *Option* to be filled in by *copy*.

copy()

Return an independent copy of the option.

is_default()

Whether the option has its default value.

reset()

Reset the option to its default value.

property value

class picos.modeling.options.**Options**(**args*, ***kwargs*)

Bases: `object`

Collection of optimization solver options.

A collection of options that affect how a *Problem* is solved. *Problem.options* is an instance of this class.

The options can be accessed as an attribute or as an item. The latter approach supports Unix shell-style wildcard characters:

```
>>> import picos
>>> P = picos.Problem()
>>> P.options.verbosity = 2
>>> P.options["primals"] = False
>>> # Set all absolute tolerances at once.
>>> P.options["abs_*_tol"] = 1e-6
```

There are two corresponding ways to reset an option to its default value:

```
>>> del P.options.verbosity
>>> P.options.reset("primals", "*_tol")
```

Options can also be passed as a keyword argument sequence when the *Problem* is created and whenever a solution is searched:

```
>>> # Use default options except for verbosity.
>>> P = picos.Problem(verbosity = 1)
>>> x = P.add_variable("x", lower = 0); P.set_objective("min", x)
>>> # Only for the next search: Don't be verbose anyway.
>>> solution = P.solve(solver = "cvxopt", verbosity = 0)
```

Available Options

Jump to option: ↪ *abs_bnb_opt_tol* ↪ *abs_dual_fsb_tol* ↪ *abs_ipm_opt_tol* ↪ *abs_prim_fsb_tol* ↪ *ad_hoc_solver* ↪ *apply_solution* ↪ *assume_conic* ↪ *cplex_bnd_monitor* ↪ *cplex_lwr_bnd_limit* ↪ *cplex_params* ↪ *cplex_upr_bnd_limit* ↪ *cplex_vmconfig* ↪ *cvxopt_kktreg* ↪ *cvxopt_kktsolver* ↪ *dualize* ↪ *duals* ↪ *gurobi_matint* ↪ *gurobi_params* ↪ *hotstart* ↪ *integrality_tol* ↪ *license_warnings* ↪ *lp_node_method* ↪ *lp_root_method* ↪ *markowitz_tol* ↪ *max_footprints* ↪ *max_fsb_nodes* ↪ *max_iterations* ↪ *mosek_basic_sol* ↪ *mosek_params* ↪ *mosek_server* ↪ *mskfsn_params* ↪ *osqp_params* ↪ *penalty_cplex* ↪ *penalty_cvxopt* ↪ *penalty_ecos* ↪ *penalty_glpk* ↪ *penalty_gurobi* ↪ *penalty_mosek* ↪ *penalty_mskfsn* ↪ *penalty_osqp* ↪ *penalty_qics* ↪ *penalty_scip* ↪ *penalty_smcp* ↪ *pool_abs_gap* ↪ *pool_rel_gap* ↪ *pool_size* ↪ *primals* ↪ *qics_params* ↪ *rel_bnb_opt_tol* ↪ *rel_dual_fsb_tol* ↪ *rel_ipm_opt_tol* ↪ *rel_prim_fsb_tol* ↪ *scip_params* ↪ *solver* ↪ *strict_options* ↪ *timelimit* ↪ *treememory* ↪ *verbosity* ↪ *verify_prediction*

Option	Default	Description
abs_bnb_opt_tol	1e-06	Absolute optimality tolerance for branch-and-bound solution strategies to mixed integer problems. A solution is optimal if the absolute difference between the objective function value of the current best integer solution and the current best bound obtained from a continuous relaxation is smaller than this value. `None` lets the solver use its own default value.
abs_dual_fsb_tol	1e-08	Absolute dual problem feasibility tolerance. A dual solution is feasible if some norm over the vector of dual constraint violations is smaller than this value. Serves as an optimality criterion for the Simplex algorithm. `None` lets the solver use its own default value.
abs_ipm_opt_tol	1e-08	Absolute optimality tolerance for interior point methods. Depending on the solver, a fesible primal/dual solution pair is considered optimal if this value upper bounds either • the absolute difference between the primal and dual objective values, or • the violation of the complementary slackness condition. The violation is computed as some norm over the vector that contains the products of each constraint's slack with its corresponding dual value. If the norm is the 1-norm, then the two conditions are equal. Otherwise they can differ by a factor that depends on the number and type of constraints. `None` lets the solver use its own default value.
abs_prim_fsb_tol	1e-08	Absolute primal problem feasibility tolerance. A primal solution is feasible if some norm over the vector of primal constraint violations is smaller than this value. `None` lets the solver use its own default value.
ad_hoc_solver	None	The solver to use as a *`Solver`* subclass. This allows solver implementations to be shipped independent of PICOS. If set, takes precedence over *solver*.
apply_solution	True	Whether to immediately apply the solution returned by a solver to the problem's variables and constraints. If multiple solutions are returned by the solver, then the first one will be applied. If this is `False`, then solutions can be applied manually via their *`apply`* method.
assume_conic	True	Determines how *`ConicQuadraticConstraint`* instances, which correspond to nonconvex constraints of the form $x^TQx + p^Tx + q \leq (a^Tx + b)(c^Tx + d)$ with $x^TQx + p^Tx + q$ representable as a squared norm, are processed: • `True` strengthens them into convex conic constraints by assuming the additional constraints $a^Tx + b \geq 0$ and $c^Tx + d \geq 0$. • `False` takes them verbatim and also considers solutions with $(a^Tx + b) < 0$ or $(c^Tx + d) < 0$. This requires a solver that accepts nonconvex quadratic constraints. **Warning:** *`ConicQuadraticConstraint`* are also used in the case of $Q = 0$. For instance, $x^2 \geq 1$ is effectively ransformed to $x \geq 1$ if this is `True`.

continues on next page

Table 1 – continued from previous page

Option	Default	Description
cplex_bnd_monitor	False	Tells CPLEX to store information about the evolution of the bounds during the MIP solution search 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`*.
cplex_lwr_bnd_limit	None	Tells CPLEX to stop MIP optimization if a lower bound below this value is found.
cplex_params	{}	A dictionary of CPLEX parameters to be set after general options are passed and before the search is started. For example, `{"mip.limits.cutpasses": 5}` limits the number of cutting plane passes when solving the root node to 5.
cplex_upr_bnd_limit	None	Tells CPLEX to stop MIP optimization if an upper bound above this value is found.
cplex_vmconfig	None	Load a CPLEX virtual machine configuration file.
cvx-opt_kktreg	1e-09	The KKT solver regularization term used by CVXOPT internally. This is an undocumented feature of CVXOPT, see here. End of 2020, this option only affected the LDL KKT solver. `None` denotes CVXOPT's default value.
cvx-opt_kktsolver	None	The KKT solver used by CVXOPT internally. See CVXOPT's guide on exploiting structure. `None` denotes PICOS' choice: Try first with the faster `"chol"`, then with the more reliable `"ldl"` solver.
dualize	False	Whether to dualize the problem as part of the solution strategy. This can sometimes lead to a significant solution search speedup.
duals	None	Whether to request a dual solution. • `True` will raise an exception if no optimal dual solution is found. • `None` will accept and apply also incomplete, infeasible or suboptimal dual solutions. • `False` will not ask for a dual solution and throw away any dual solution returned by the solver.
gurobi_matint	None	Whether to use Gurobi's matrix interface. This requires Gurobi 9 or later and SciPy. `None` with *`PREFER_GUROBI_MATRIX_INTERFACE`* enabled means *use it if possible*. `None` with that setting disabled behaves like `False`.
gurobi_params	{}	A dictionary of Gurobi parameters to be set after general options are passed and before the search is started. For example, `{"NodeLimit": 25}` limits the number of nodes visited by the MIP optimizer to 25.
hotstart	False	Tells the solver to start from the (partial) solution that is stored in the *`variables`* assigned to the problem.
integrality_tol	1e-05	Integrality tolerance. A number $x \in \mathbb{R}$ is considered integral if $\min_{z\in\mathbb{Z}} \lvert x - z\rvert$ is at most this value. `None` lets the solver use its own default value.
license_warnings	True	Whether solvers are allowed to ignore the *verbosity* option to print licensing related warnings. See also the global setting *`LICENSE_WARNINGS`*.

continues on next page

Table 1 – continued from previous page

Option	Default	Description
lp_node_method	None	Algorithm used to solve continuous linear problems at non-root nodes of the branching tree built when solving mixed integer programs. • None lets PICOS or the solver select it for you. • "psimplex" for Primal Simplex. • "dsimplex" for Dual Simplex. • "interior" for the interior point method.
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 for you. • "psimplex" for Primal Simplex. • "dsimplex" for Dual Simplex. • "interior" for the interior point method.
markowitz_tol	None	Markowitz threshold used in the Simplex algorithm. None lets the solver use its own default value.
max_footprints	1024	Maximum number of different predicted problem formulations (footprints) to consider before deciding on a formulation and solver to use. None lets PICOS exhaust all reachable problem formulations.
max_fsb_nodes	None	Maximum number of feasible solution nodes visited for branch-and-bound solution strategies. None means no limit. **Note:** If you want to obtain all feasible solutions that the solver encountered, use the *pool_size* option.
max_iterations	None	Maximum number of iterations allowed for iterative solution strategies. None means no limit.
mosek_basic_sol	False	Return a basic solution when solving LPs with MOSEK (Optimizer).
mosek_params	{}	A dictionary of MOSEK (Optimizer) parameters to be set after general options are passed and before the search is started. See the list of MOSEK (Optimizer) 8.1 parameters.
mosek_server	None	Address of a MOSEK remote optimization server to use. This option affects both MOSEK (Optimizer) and MOSEK (Fusion).
msk-fsn_params	{}	A dictionary of MOSEK (Fusion) parameters to be set after general options are passed and before the search is started. See the list of MOSEK (Fusion) 8.1 parameters.
osqp_params	{}	A dictionary of OSQP parameters to be set after general options are passed and before the search is started. See the list of OSQP parameters.
penalty_cplex	0.0	Penalty for using the CPLEX solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.

continues on next page

Table 1 – continued from previous page

Option	Default	Description
penalty_cvxopt	1.0	Penalty for using the CVXOPT solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_ecos	1.0	Penalty for using the ECOS solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_glpk	1.0	Penalty for using the GLPK solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_gurobi	0.0	Penalty for using the GUROBI solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_mosek	0.0	Penalty for using the MOSEK solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_mskfsn	0.5	Penalty for using the MSKFSN solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_osqp	2.0	Penalty for using the OSQP solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_qics	1.0	Penalty for using the QICS solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_scip	1.0	Penalty for using the SCIP solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
penalty_smcp	2.0	Penalty for using the SMCP solver. If solver A has a penalty of p and solver B has a larger penality of $p + x$, then B is be chosen over A only if the problem as passed to A would be 10^x times larger as when passed to B.
pool_abs_gap	None	Discards solutions from the *solution pool* as soon as a better solution is found that beats it by the given absolute objective function gap. `None` is the solver's choice, which may be *never discard*.

continues on next page

Table 1 – continued from previous page

Option	Default	Description
pool_rel_gap	None	Discards solutions from the *solution pool* as soon as a better solution is found that beats it by the given relative objective function gap. None is the solver's choice, which may be *never discard*.
pool_size	None	Maximum number of mixed integer feasible solutions returned. If this is not None, `solve` returns a list of `Solution` objects instead of just a single one. None lets the solver return only the best solution.
primals	True	Whether to request a primal solution. • True will raise an exception if no optimal primal solution is found. • None will accept and apply also incomplete, infeasible or suboptimal primal solutions. • False will not ask for a primal solution and throw away any primal solution returned by the solver.
qics_params	{}	A dictionary of QICS parameters to be set after general options are passed and before the search is started. See the list of QICS parameters.
rel_bnb_opt_tol	0.0001	Relative optimality tolerance for branch-and-bound solution strategies to mixed integer problems. Like *abs_bnb_opt_tol*, but the difference is divided by a convex combination of the absolute values of the two objective function values. None lets the solver use its own default value.
rel_dual_fsb_tol	1e-08	Relative dual problem feasibility tolerance. Like *abs_dual_fsb_tol*, but the norm is divided by the norm of the constraints' right hand side vector. (See *rel_prim_fsb_tol* for exceptions.) Serves as an optimality criterion for the Simplex algorithm. None lets the solver use its own default value.
rel_ipm_opt_tol	1e-08	Relative optimality tolerance for interior point methods. Like *abs_ipm_opt_tol*, but the suboptimality measure is divided by a convex combination of the absolute primal and dual objective function values. None lets the solver use its own default value.
rel_prim_fsb_tol	1e-08	Relative primal problem feasibility tolerance. Like *abs_prim_fsb_tol*, but the norm is divided by the norm of the constraints' right hand side vector. If the norm used is some nested norm (e.g. the maximum over the norms of the equality and inequality violations), then solvers might divide the inner violation norms by the respective right hand side inner norms (see e.g. CVXOPT). To prevent that the right hand side vector norm is zero (or small), solvers would either add some small constant or use a fixed lower bound, which may be as large as 1. None lets the solver use its own default value.
scip_params	{}	A dictionary of SCIP parameters to be set after general options are passed and before the search is started. For example, `{"lp/threads": 4}` sets the number of threads to solve LPs with to 4.

continues on next page

Table 1 – continued from previous page

Option	Default	Description
solver	None	The solver to use. See also the global settings *SOLVER_BLACKLIST*, *SOLVER_WHITELIST* and *NONFREE_SOLVERS*. • None to let PICOS choose. • "cplex" for *CPLEXSolver*. • "cvxopt" for *CVXOPTSolver*. • "ecos" for *ECOSSolver*. • "glpk" for *GLPKSolver*. • "gurobi" for *GurobiSolver*. • "mosek" for *MOSEKSolver*. • "mskfsn" for *MOSEKFusionSolver*. • "osqp" for *OSQPSolver*. • "qics" for *QICSSolver*. • "scip" for *SCIPSolver*. • "smcp" for *SMCPSolver*. This option is ignored when *ad_hoc_solver* is set. **Note:** *picos.available_solvers()* returns a list of names of solvers that are available at runtime.
strict_options	False	Whether unsupported general options will raise an *UnsupportedOptionError* exception, instead of printing a warning.
timelimit	None	Maximum number of seconds spent searching for a solution. None means no limit.
treememory	None	Bound on the memory used by the branch-and-bound tree, in Megabytes. None means no limit.
verbosity	0	Verbosity level. • -1 attempts to suppress all output, even erros. • 0 only generates warnings and errors. • 1 generates standard informative output. • 2 or larger prints additional information for debugging purposes.
verify_prediction	True	Whether PICOS should validate that problem reformulations produce a problem that matches their predicted outcome. If a mismatch is detected, a RuntimeError is thrown as there is a chance that it is caused by a bug in the reformulation, which could affect the correctness of the solution. By disabling this option you are able to retrieve a correct solution given that the error is only in the prediction, and given that the solution strategy remains valid for the actual outcome.

__eq__(*other*)

Report whether two sets of options are equal.

__init__(***options*)

Create a default option set and set the given options on top.

static __new__(*cls*, **args*, ***kwargs*)

Create an empty options set.

copy()

Return an independent copy of the current options set.

help(**options*)

Print text describing selected options.

Parameters
options – The names of the options to describe, may contain wildcard characters.

reset(**options*)

Reset all or a selection of options to their default values.

Parameters
options – The names of the options to reset, may contain wildcard characters. If no name is given, all options are reset.

self_or_updated(***options*)

Return either a modified copy or self, depending on given options.

update(***options*)

Set multiple options at once.

This method is called with the keyword arguments supplied to the `Options` constructor, so the following two are the same:

```
>>> import picos
>>> a = picos.Options(verbosity = 1, primals = False)
>>> b = picos.Options()
>>> b.update(verbosity = 1, primals = False)
>>> a == b
True
```

Parameters
options – A parameter sequence of options to set.

updated(***options*)

Return a modified copy.

property nondefaults

A dictionary mapping option names to nondefault values.

Example

```
>>> from picos import Options
>>> o = Options()
>>> o.verbosity = 2
>>> o.nondefaults
{'verbosity': 2}
>>> Options(**o.nondefaults) == o
True
```

Objects

picos.modeling.options.**OPTIONS**

The table of available solver options.

Each entry is a tuple representing a single solver option. The tuple's entries are, in order:

- Name of the option. Must be a valid Python attribute name.
- The option's argument type. Will be cast on any argument that is not already an instance of the type, except for `None`.
- The option's default value. Must already be of the proper type, or `None`, and must pass the optional check.
- The option's description, which is used as part of the docstring of *`Options`*. In the case of a multi-line text, leading and trailing empty lines as well as the overall indentation are ignored.
- Optional: A boolean function used on every argument that passes the type conversion (so either an argument of the proper type, or `None`). If the function returns `False`, then the argument is rejected. The default function rejects exactly `None`. Supplying `None` instead of a function accepts all arguments (in particular, accepts `None`).

Default value

```
[('abs_bnb_opt_tol',
  <class 'float'>,
  1e-06,
  '\n'
  '         Absolute optimality tolerance for branch-and-bound␣
↪solution '
  'strategies\n'
  '         to mixed integer problems.\n'
  '\n'
  '         A solution is optimal if the absolute...
```

picos.modeling.options.**OPTION_OBJS**

The initial solver options as *`Option`* objects.

Default value

```
(<picos.modeling.options.Option object at 0x7d37765cb8f0>,
 <picos.modeling.options.Option object at 0x7d37765cb8c0>,
 <picos.modeling.options.Option object at 0x7d37765cb5f0>,
 <picos.modeling.options.Option object at 0x7d37765cb5c0>,
...
```

picos.modeling.problem

Implementation of *`Problem`*.

Exceptions

exception picos.modeling.problem.**SolutionFailure**(*code*, *message*)

Bases: RuntimeError

Solving the problem failed.

Classes

class picos.modeling.problem.**Problem**(*name=None*, ***, *copyOptions=None*, *useOptions=None*, ***extra_options*)

Bases: *Valuable*

PICOS' representation of an optimization problem.

Example

```
>>> from picos import Problem, RealVariable
>>> X = RealVariable("X", (2,2), lower = 0)
>>> P = Problem("Example")
>>> P.maximize = X.tr
>>> C = X.sum <= 10
>>> P += C, X[0,0] == 1
>>> print(P)
Example (Linear Program)
  maximize tr(X)
  over
    2×2 real variable X (bounded below)
  subject to
    ∑(X) ≤ 10
    X[0,0] = 1
>>> # PICOS will select a suitable solver if you don't specify one.
>>> solution = P.solve(solver = "cvxopt")
>>> solution.claimedStatus
'optimal'
>>> solution.searchTime
0.002137422561645508
>>> round(P, 1)
10.0
>>> print(X)
[ 1.00e+00  4.89e-10]
[ 4.89e-10  9.00e+00]
>>> round(C.dual, 1)
1.0
```

__iadd__(*constraints*)

See require.

__init__(*name=None*, ***, *copyOptions=None*, *useOptions=None*, ***extra_options*)

Create an empty problem and optionally set initial solver options.

Parameters

- **name** (*str*) – A name or title to give to the optimization problem.
- **copyOptions** – An *Options* object to copy instead of using the default options.
- **useOptions** – An *Options* object to use (without making a copy) instead of using the default options.

- **extra_options** – A sequence of additional solver options to apply on top of the default options or those given by `copyOptions` or `useOptions`.

add_constraint(*constraint*, *key=None*)

Add a single constraint to the problem and return it.

Parameters

- **constraint** (*Constraint*) – The constraint to be added.
- **key** – DEPRECATED

Returns

The constraint that was added to the problem.

Note: This method is superseded by the more compact and more flexible *require* method or, at your preference, the += operator.

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

Add constraints from an iterable to the problem.

Parameters

- **lst** – Iterable of constraints to add.
- **it** – DEPRECATED
- **indices** – DEPRECATED
- **key** – DEPRECATED

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 Real Variable: x[0]>,
 <2×1 Real Variable: x[1]>,
 <2×1 Real Variable: x[2]>,
 <2×1 Real Variable: x[3]>,
 <2×1 Real 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)
Feasibility Problem
  find an assignment
  for
    2×1 real variable x[i] ∀ i ∈ [0...4]
```

(continues on next page)

(continued from previous page)

```
    3×1 real variable w[i,j] ∀ (i,j) ∈ zip([1,2,4],[2,0,2])
    5×1 real variable y
  subject to
    uᵀ·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]
```

Note: This method is superseded by the more compact and more flexible `require` method or, at your preference, the += operator.

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

Legacy method to create a PICOS variable.

Parameters

- **name** (*str*) – The name of the variable.
- **size** (anything recognized by *load_shape*) – The shape of the variable.
- **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' or 'skewsym' – skew-symmetric matrix,
  - 'complex' – complex matrix,
  - 'hermitian' – complex hermitian matrix.
- **lower** (anything recognized by *load_data*) – A lower bound on the variable.
- **upper** (anything recognized by *load_data*) – An upper bound on the variable.

Returns

A *BaseVariable* instance.

Example

```
>>> from picos import Problem
>>> P = Problem()
>>> x = P.add_variable("x", 3)
>>> x
<3×1 Real Variable: x>
>>> # Variable are not stored inside the problem any more:
>>> P.variables
mappingproxy(OrderedDict())
>>> # They are only part of the problem if they actually appear:
>>> P.set_objective("min", abs(x)**2)
>>> P.variables
mappingproxy(OrderedDict({'x': <3×1 Real Variable: x>}))
```

Deprecated since version 2.0: Variables can now be created independent of problems, and do not need to be added to any problem explicitly.

as_dual()

Return the Lagrangian dual problem of the standardized problem.

Deprecated since version 2.0: Use *dual* instead.

check_current_value_feasibility(*tol=1e-5*, *inttol=None*)

Check if the problem is feasibly valued.

Checks whether all variables that appear in constraints are valued and satisfy both their bounds and the constraints up to the given tolerance.

Parameters

- **tol** (*float*) – Largest tolerated absolute violation of a constraint or variable bound. If `None`, then the `abs_prim_fsb_tol` solver option is used.
- **inttol** – DEPRECATED

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.

Raises

picos.uncertain.IntractableWorstCase – When computing the worst-case (expected) value of the constrained expression is not supported.

clone(*copyOptions=True*)

Create a semi-deep copy of the problem.

The copy is constrained by the same constraint objects and has the same objective function and thereby references the existing variables and parameters that appear in these objects.

The clone can be modified to describe a new problem but when its variables and parameters are valued, in particular when a solution is applied to the new problem, then the same values are found in the corresponding variables and parameters of the old problem. If this is not a problem to you, then cloning can be much faster than copying.

Parameters

copyOptions (*bool*) – Whether to make an independent copy of the problem's options. Disabling this will apply any option changes to the original problem as well but yields a (very small) reduction in cloning time.

continuous_relaxation(*copy_other_mutables=True*)

Return a continuous relaxation of the problem.

This is done by replacing integer variables with continuous ones.

Parameters

copy_other_mutables (*bool*) – Whether variables that are already continuous as well as parameters should be copied. If this is `False`, then the relxation shares these mutables with the original problem.

copy()

Create a deep copy of the problem, using new mutables.

get_constraint(*idOrIndOrCon*)

Return a (list of) constraint(s) of the problem.

Parameters

idOrIndOrCon (`picos.constraints.Constraint` *or* *int* *or* *tuple* *or* *list*) – One of the following:

- A constraint object. It will be returned when the constraint is part of the problem, otherwise a KeyError is raised.
- The integer ID of the constraint.
- The integer offset of the constraint in the list of all constraints that are part of the problem, in the order that they were added.

- A list or tuple of length 1. Its only element is the index of a constraint group (of constraints that were added together), where groups are indexed in the order that they were added to the problem. The whole group is returned as a list of constraints. That list has the constraints in the order that they were added.
- A list or tuple of length 2. The first element is a constraint group offset as above, the second an offset within that 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([x[i].sum < y[i] for i in range(5)])
>>> Cy=prob.add_constraint(y>0)
>>> print(prob)
Linear Feasibility Problem
  find an assignment
  for
    2×1 real variable x[i] ∀ i ∈ [0...4]
    5×1 real variable y
  subject to
    ∑(x[i]) ≤ y[i] ∀ i ∈ [0...4]
    y ≥ 0
>>> # Retrieve the second constraint, indexed from zero:
>>> prob.get_constraint(1)
<1×1 Affine Constraint: ∑(x[1]) ≤ y[1]>
>>> # Retrieve the fourth consraint from the first group:
>>> prob.get_constraint((0,3))
<1×1 Affine Constraint: ∑(x[3]) ≤ y[3]>
>>> # Retrieve the whole first group of constraints:
>>> pprint(prob.get_constraint((0,)))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[1]) ≤ y[1]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(x[3]) ≤ y[3]>,
 <1×1 Affine Constraint: ∑(x[4]) ≤ y[4]>]
>>> # Retrieve the second "group", containing just one constraint:
>>> prob.get_constraint((1,))
[<5×1 Affine Constraint: y ≥ 0>]
```

get_valued_variable(*name*)

Retrieve values of variables referenced by the problem.

This method works the same `get_variable` but it returns the variable's `values` instead of the variable objects.

Raises
`NotValued` – If any of the selected variables is not valued.

get_variable(*name*)

Retrieve variables referenced by the problem.

Retrieves either a single variable with the given name or a group of variables all named `name[param]` with different values for `param`. If the values for `param` are the integers from zero to the size of the group minus one, then the group is returned as a `list` ordered by `param`, otherwise it is returned as a `dict` with the values of `param` as keys.

Note: Since PICOS 2.0, variables are independent of problems and only appear in a problem for as long as they are referenced by the problem's objective function or constraints.

Parameters
name (*str*) – The name of a variable, or the base name of a group of variables.

Returns
A *variable* or a `list` or `dict` thereof.

Example

```
>>> from picos import Problem, RealVariable
>>> from pprint import pprint
>>> # Create a number of variables with structured names.
>>> vars = [RealVariable("x")]
>>> for i in range(4):
...     vars.append(RealVariable("y[{}]".format(i)))
>>> for key in ["alice", "bob", "carol"]:
...     vars.append(RealVariable("z[{}]".format(key)))
>>> # Make the variables appear in a problem.
>>> P = Problem()
>>> P.set_objective("min", sum([var for var in vars]))
>>> print(P)
Linear Program
  minimize x + y[0] + y[1] + y[2] + y[3] + z[alice] + z[bob] + z[carol]
  over
    1×1 real variables x, y[0], y[1], y[2], y[3], z[alice], z[bob],
      z[carol]
>>> # Retrieve the variables from the problem.
>>> P.get_variable("x")
<1×1 Real Variable: x>
>>> pprint(P.get_variable("y"))
[<1×1 Real Variable: y[0]>,
 <1×1 Real Variable: y[1]>,
 <1×1 Real Variable: y[2]>,
 <1×1 Real Variable: y[3]>]
>>> pprint(P.get_variable("z"))
{'alice': <1×1 Real Variable: z[alice]>,
 'bob': <1×1 Real Variable: z[bob]>,
 'carol': <1×1 Real Variable: z[carol]>}
>>> P.get_variable("z")["alice"] is P.get_variable("z[alice]")
True
```

is_continuous()

Whether all variables are of continuous types.

Deprecated since version 2.0: Use *continuous* instead.

is_pure_integer()

Whether all variables are of integral types.

Deprecated since version 2.0: Use *pure_integer* instead.

obj_value()

Objective function value.

Raises

`AttributeError` – If the problem is a feasibility problem or if the objective function is not valued. This is legacy behavior. Note that `value` just returns `None` while functions that **do** raise an exception to denote an unvalued expression would raise *`NotValued`* instead.

Deprecated since version 2.0: Use `value` instead.

`prepared`(*steps=None*, ***extra_options*)

Perform a dry-run returning the reformulated (prepared) problem.

This behaves like *`solve`* in that it takes a number of additional temporary options, finds a solution strategy matching the problem and options, and performs the strategy's reformulations in turn to obtain modified problems. However, it stops after the given number of steps and never hands the reformulated problem to a solver. Instead of a solution, *`prepared`* then returns the last reformulated problem.

Unless this method returns the problem itself, the special attributes `prepared_strategy` and `prepared_steps` are added to the returned problem. They then contain the (partially) executed solution strategy and the number of performed reformulations, respectively.

Parameters

- **`steps`** (*`int`*) – Number of reformulations to perform. `None` means as many as there are. If this parameter is 0, then the problem itself is returned. If it is 1, then only the implicit first reformulation *`ExtraOptions`* is executed, which may also output the problem itself, depending on `extra_options`.
- **`extra_options`** – Additional solver options to use with this dry-run only.

Returns

The reformulated problem, with `extra_options` set unless they were "consumed" by a reformulation (e.g. *option_dualize*).

Raises

- *`NoStrategyFound`* – If no solution strategy was found.
- `ValueError` – If there are not as many reformulation steps as requested.

Example

```
>>> from picos import Problem, RealVariable
>>> x = RealVariable("x", 2)
>>> P = Problem()
>>> P.set_objective("min", abs(x)**2)
>>> Q = P.prepared(solver = "cvxopt")
>>> print(Q.prepared_strategy)  # Show prepared reformulation steps.
1. ExtraOptions
2. EpigraphReformulation
3. SquaredNormToConicReformulation
4. CVXOPTSolver
>>> Q.prepared_steps  # Check how many steps have been performed.
3
>>> print(P)
Quadratic Program
  minimize ‖x‖²
  over
    2×1 real variable x
>>> print(Q)
Second Order Cone Program
  minimize __..._t
  over
    1×1 real variable __..._t
```

(continues on next page)

(continued from previous page)

```
    2×1 real variable x
  subject to
    ‖fullroot(‖x‖²)‖² ≤  __..._t ∨ __..._t ≥  0
```

reformulated(*specification*, ***extra_options*)

Return the problem reformulated to match a specification.

Internally this creates a dummy solver accepting problems of the desired form and then calls `prepared` with the dummy solver passed via *option_ad_hoc_solver*. See meth:*prepared* for more details.

Parameters

- **specification** (`Specification`) – A problem class that the resulting problem must be a member of.
- **extra_options** – Additional solver options to use with this reformulation only.

Returns

The reformulated problem, with `extra_options` set unless they were "consumed" by a reformulation (e.g. *dualize*).

Raises

NoStrategyFound – If no reformulation strategy was found.

Example

```
>>> from picos import Problem, RealVariable
>>> from picos.modeling import Specification
>>> from picos.expressions import AffineExpression
>>> from picos.constraints import (
...     AffineConstraint, SOCConstraint, RSOCConstraint)
>>> # Define the class/specification of second order conic problems:
>>> S = Specification(objectives=[AffineExpression],
...     constraints=[AffineConstraint, SOCConstraint, RSOCConstraint])
>>> # Define a quadratic program and reformulate it:
>>> x = RealVariable("x", 2)
>>> P = Problem()
>>> P.set_objective("min", abs(x)**2)
>>> Q = P.reformulated(S)
>>> print(P)
Quadratic Program
  minimize ‖x‖²
  over
    2×1 real variable x
>>> print(Q)
Second Order Cone Program
  minimize __..._t
  over
    1×1 real variable __..._t
    2×1 real variable x
  subject to
    ‖fullroot(‖x‖²)‖² ≤  __..._t ∨ __..._t ≥  0
```

Note: This method is intended for educational purposes. You do not need to use it when solving a problem as PICOS will perform the necessary reformulations automatically.

remove_all_constraints()

Remove all constraints from the problem.

Note: This method does not remove bounds set directly on variables.

remove_constraint(*idOrIndOrCon*)

Delete a constraint from the problem.

Parameters

idOrIndOrCon – See *get_constraint*.

Example

```
>>> import picos
>>> from pprint import pprint
>>> P = picos.Problem()
>>> x = [P.add_variable('x[{0}]'.format(i), 2) for i in range(4)]
>>> y = P.add_variable('y', 4)
>>> Cxy = P.add_list_of_constraints(
...     [x[i].sum <= y[i] for i in range(4)])
>>> Cy = P.add_constraint(y >= 0)
>>> Cx0to2 = P.add_list_of_constraints([x[i] <= 2 for i in range(3)])
>>> Cx3 = P.add_constraint(x[3] <= 1)
>>> pprint(list(P.constraints.values()))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[1]) ≤ y[1]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(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):
>>> P.remove_constraint(1)
>>> pprint(list(P.constraints.values()))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(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:
>>> P.remove_constraint((1,))
>>> pprint(list(P.constraints.values()))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(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]:
>>> P.remove_constraint((2,))
>>> pprint(list(P.constraints.values()))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(x[3]) ≤ y[3]>,
```

(continues on next page)

(continued from previous page)

```
 <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|:
>>> P.remove_constraint((1,1))
>>> pprint(list(P.constraints.values()))
[<1×1 Affine Constraint: ∑(x[0]) ≤ y[0]>,
 <1×1 Affine Constraint: ∑(x[2]) ≤ y[2]>,
 <1×1 Affine Constraint: ∑(x[3]) ≤ y[3]>,
 <2×1 Affine Constraint: x[0] ≤ [2]>,
 <2×1 Affine Constraint: x[2] ≤ [2]>]
```

remove_variable(*name*)

Does nothing.

Deprecated since version 2.0: Whether a problem references a variable is now determined dynamically, so this method has no effect.

require(**constraints*, *ret=False*)

Add constraints to the problem.

Parameters

- **constraints** – A sequence of constraints or constraint groups (iterables yielding constraints) or a mix thereof.
- **ret** (*bool*) – Whether to return the added constraints.

Returns

When `ret=True`, returns either the single constraint that was added, the single group of constraint that was added in the form of a `list` or, when multiple arguments are given, a list of constraints or constraint groups represented as above. When `ret=False`, returns nothing.

Example

```
>>> from picos import Problem, RealVariable
>>> x = RealVariable("x", 5)
>>> P = Problem()
>>> P.require(x >= -1, x <= 1)  # Add individual constraints.
>>> P.require([x[i] <= x[i+1] for i in range(4)])  # Add groups.
>>> print(P)
Linear Feasibility Problem
  find an assignment
  for
    5×1 real variable x
  subject to
    x ≥ [-1]
    x ≤ [1]
    x[i] ≤ x[i+1] ∀ i ∈ [0…3]
```

Note: For a single constraint `C`, `P.require(C)` may also be written as `P += C`. For multiple constraints, `P.require([C1, C2])` can be abbreviated `P += [C1, C2]` while `P.require(C1, C2)` can be written as either `P += (C1, C2)` or just `P += C1, C2`.

reset(*resetOptions=False*)

Reset the problem instance to its initial empty state.

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

`set_all_options_to_default()`

Set all solver options to their default value.

Deprecated since version 2.0: Use `Problem.options` instead.

`set_objective`(*direction=None*, *expression=None*)

Set the optimization direction and objective function of the problem.

Parameters

- `direction` (*str*) – Case insensitive search direction string. One of
  - `"min"` or `"minimize"`,
  - `"max"` or `"maximize"`,
  - `"find"` or `None` (for a feasibility problem).
- `expression` (`Expression`) – The objective function. Must be `None` for a feasibility problem.

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

Set 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.

Deprecated since version 2.0: Use `Problem.options` instead.

`set_var_value`(*name*, *value*)

Set the `value` of a variable.

For a `Problem` `P`, this is the same as `P.variables[name] = value`.

Parameters

- `name` (*str*) – Name of the variable to be valued.
- `value` (anything recognized by `load_data`) – The value to be set.

Deprecated since version 2.0: Use `variables` instead.

`solve`(***extra_options*)

Hand 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.

The default behavior (options `primals=True`, `duals=None`) is to raise a `SolutionFailure` when the primal solution is not found optimal by the solver, while the dual solution is allowed to be missing or incomplete.

When this method succeeds and unless `apply_solution=False`, you can access the solution as follows:

- The problem's `value` denotes the objective function value.
- The variables' `value` is set according to the primal solution. You can in fact query the value of any expression involving valued variables like this.
- The constraints' `dual` is set according to the dual solution.
- The value of any parameter involved in the problem may have changed, depending on the parameter.

Parameters

`extra_options` – A sequence of additional solver options to use with this solution search only. In particular, this lets you

- select a solver via the `solver` option,
- obtain non-optimal primal solutions by setting `primals=None`,
- require a complete and optimal dual solution with `duals=True`, and
- skip valuing variables or constraints with `apply_solution=False`.

Returns ~picos.Solution or list(~picos.Solution)

A solution object or list thereof.

Raises

`SolutionFailure` – In the following cases:

1. No solution strategy was found.
2. Multiple solutions were requested but none were returned.
3. A primal solution was explicitly requested (`primals=True`) but the primal solution is missing/incomplete or not claimed optimal.
4. A dual solution was explicitly requested (`duals=True`) but the dual solution is missing/incomplete or not claimed optimal.

The case number is stored in the `code` attribute of the exception.

`update_options`(***options*)

Set multiple solver options at once.

Parameters

`options` – A parameter sequence of options to set.

Deprecated since version 2.0: Use *`Problem.options`* instead.

`verbosity`()

Return the problem's current verbosity level.

Deprecated since version 2.0: Use *`Problem.options`* instead.

`write_to_file`(*filename*, *writer='picos'*)

See *`picos.modeling.file_out.write`*.

`CONIC_FORM = <Specification: Optimize AffineExpression subject to AffineConstraint, ComplexAffineConstraint, ComplexLMIConstraint, DummyConstraint, LMIConstraint, ProductConeConstraint, RSOCConstraint, SOCConstraint using any variables and any options.>`

The specification for problems returned by *`conic_form`*.

`property conic_form`

The problem in conic form.

Reformulates the problem such that the objective is affine and all constraints are *`ConicConstraint`* instances.

Raises

`NoStrategyFound` – If no reformulation strategy was found.

Example

```
>>> from picos import Problem, RealVariable
>>> x = RealVariable("x", 2)
>>> P = Problem()
>>> P.set_objective("min", abs(x)**2)
```

(continues on next page)

(continued from previous page)

```
>>> print(P)
Quadratic Program
  minimize ‖x‖²
  over
    2×1 real variable x
>>> print(P.conic_form)
Second Order Cone Program
  minimize __..._t
  over
    1×1 real variable __..._t
    2×1 real variable x
  subject to
    ‖fullroot(‖x‖²)‖² ≤ __..._t ∨ __..._t ≥ 0
```

Note: This property is intended for educational purposes. You do not need to use it when solving a problem as PICOS will perform the necessary reformulations automatically.

property constraints

Maps constraint IDs to constraints that are part of the problem.

Returns
A read-only view to an `OrderedDict`. The order is that in which constraints were added.

property continuous

Whether all variables are of continuous types.

property countCons

The same as `len` applied to *`constraints`*.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property countVar

The same as `len` applied to *`variables`*.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property dual

The Lagrangian dual problem of the standardized problem.

More precisely, this property invokes the following:

1. The primal problem is posed as an equivalent conic standard form minimization problem, with variable bounds expressed as additional constraints.
2. The Lagrangian dual problem of the reposed primal is computed.
3. The optimization direction and objective function sign of the dual are adjusted such that, given strong duality and primal feasibility, the optimal values of both problems are equal. In particular, if the primal problem is a minimization or a maximization problem, the dual problem returned will be the respective other.

Raises
`NoStrategyFound` – If no reformulation strategy was found.

Note: This property is intended for educational purposes. If you want to solve the primal problem via its dual, use the *dualize* option instead.

property footprint

Problem footprint as a *Footprint* object.

property last_solution

The last *Solution* applied to the problem.

property maximize

Maximization objective as an *Expression*.

This can be used to set a maximization objective. For querying the objective, it is recommended to use *objective* instead.

property minimize

Minimization objective as an *Expression*.

This can be used to set a minimization objective. For querying the objective, it is recommended to use *objective* instead.

property mutables

Maps names to variables and parameters in use by the problem.

Returns

A read-only view to an `OrderedDict`. The order is deterministic and depends on the order of operations performed on the *Problem* instance as well as on the mutables' names.

property name

Name or title of the problem.

property no

Normalized objective as an *Objective* instance.

Either a minimization or a maximization objective, with feasibility posed as "minimize 0".

The same as the *normalized* attribute of the *objective*.

property numberConeConstraints

Number of quadratic conic constraints stored.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property numberLSEConstraints

Number of *LogSumExpConstraint* stored.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property numberOfVars

The sum of the dimensions of all referenced variables.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property numberQuadConstraints

Number of quadratic constraints stored.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property numberSDPConstraints

Number of *LMIConstraint* stored.

Deprecated since version 2.0: Still used internally by legacy code; will be removed together with that code.

property objective

Optimization objective as an *Objective* instance.

property options

Solution search parameters as an *Options* object.

property parameters

Maps names to parameters in use by the problem.

Returns
See *mutables*.

property pure_integer

Whether all variables are of integral types.

property status

The solution status string as claimed by *last_solution*.

property strategy

Solution strategy as a *Strategy* object.

A strategy is available once you order the problem to be solved and it will be reused for successive solution attempts (of a modified problem) while it remains valid with respect to the problem's *footprint*.

When a strategy is reused, modifications to the objective and constraints of a problem are passed step by step through the strategy's reformulation pipeline while existing reformulation work is not repeated. If the solver also supports these kinds of updates, then modifying and re-solving a problem can be much faster than solving the problem from scratch.

Example

```
>>> from picos import Problem, RealVariable
>>> x = RealVariable("x", 2)
>>> P = Problem()
>>> P.set_objective("min", abs(x)**2)
>>> print(P.strategy)
None
>>> sol = P.solve(solver = "cvxopt")  # Creates a solution strategy.
>>> print(P.strategy)
1. ExtraOptions
2. EpigraphReformulation
3. SquaredNormToConicReformulation
4. CVXOPTSolver
>>> # Add another constraint handled by SquaredNormToConicReformulation:
>>> P.add_constraint(abs(x - 2)**2 <= 1)
<Squared Norm Constraint: ‖x - [2]‖² ≤ 1>
>>> P.strategy.valid(solver = "cvxopt")
True
>>> P.strategy.valid(solver = "glpk")
False
>>> sol = P.solve(solver = "cvxopt")  # Reuses the strategy.
```

It's also possible to create a startegy from scratch:

```
>>> from picos.modeling import Strategy
>>> from picos.reforms import (EpigraphReformulation,
...     ConvexQuadraticToConicReformulation)
>>> from picos.solvers import CVXOPTSolver
>>> # Mimic what solve() does when no strategy exists:
>>> P.strategy = Strategy(P, CVXOPTSolver, EpigraphReformulation,
...     ConvexQuadraticToConicReformulation)
```

property type

The problem type as a string, such as "Linear Program".

property variables

Maps names to variables in use by the problem.

Returns
See *mutables*.

picos.modeling.quicksolve

Functions to quickly solve a problem.

Functions

picos.modeling.quicksolve.**find_assignment**(*subject_to=[]*, ***options*)

Find a feasible variable assignment.

Internally, this creates a *Problem*, *adds constraints* and performs a *solution search*.

Parameters

- **subject_to** (*list*(Constraint)) – A collection of constraints to obey.
- **options** – A keyword argument sequence of solver options to use. See *Options*.

Returns
Nothing. Check the concerned variables' *values*.

Raises
SolutionFailure – See *solve*.

Example

```
>>> from picos import find_assignment, RealVariable
>>> x = RealVariable("x")
>>> values = find_assignment([x**2 + 1 <= x], solver="cvxopt")
...
Traceback (most recent call last):
    ...
picos.modeling.problem.SolutionFailure: Code 3: ...
>>> x.value
>>> find_assignment([x**2 + 0.25 <= x], solver="cvxopt")
>>> round(x.value, 5)
0.5
```

picos.modeling.quicksolve.**maximize**(*function*, *subject_to=[]*, ***options*)

Maximize a scalar expression subject to constraints.

Internally, this creates a *Problem*, *sets an objective*, *adds constraints*, performs a *solution search* and returns an optimum value found.

Parameters

- **function** (Expression) – The objective function to maximize.
- **subject_to** (*list*(Constraint)) – A collection of constraints to obey.
- **options** – A keyword argument sequence of solver options to use. See *Options*.

Returns
The optimum value, as computed from an applied solution.

Raises
SolutionFailure – See *solve*.

Example

```
>>> from picos import maximize, RealVariable
>>> x = RealVariable("x")
>>> p = maximize(-x**2, [(x - 2)**2 <= x - 2], solver="cvxopt")
>>> round(p, 5)
-4.0
>>> round(x, 5)
2.0
```

picos.modeling.quicksolve.**minimize**(*function*, *subject_to=[]*, ***options*)

Minimize a scalar expression subject to constraints.

Internally, this creates a *Problem*, *sets an objective*, *adds constraints*, performs a *solution search* and returns an optimum value found.

Parameters

- **function** (Expression) – The objective function to minimize.
- **subject_to** (*list*(Constraint)) – A collection of constraints to obey.
- **options** – A keyword argument sequence of solver options to use. See *Options*.

Returns
The optimum value, as computed from an applied solution.

Raises
SolutionFailure – See *solve*.

Example

```
>>> from picos import minimize, RealVariable
>>> x = RealVariable("x")
>>> p = minimize(x**2, [(x - 2)**2 <= x - 2], solver="cvxopt")
>>> round(p, 5)
4.0
>>> round(x, 5)
2.0
```

picos.modeling.solution

Optimization problem solution representation.

Classes

class picos.modeling.solution.**Solution**(*primals*, *duals=None*, *problem=None*, *solver='user'*, *primalStatus=SS_UNKNOWN*, *dualStatus=SS_UNKNOWN*, *problemStatus=PS_UNKNOWN*, *searchTime=0.0*, *info=None*, *vectorizedPrimals=False*, *reportedValue=None*)

Bases: object

Assignment of primal and dual values to variables and constraints.

Instances are usually returned by a solver (and thus bound to a *problem* instance), but may be manually created by the user:

```
>>> import picos
>>> x = picos.RealVariable("x")
>>> s = picos.Solution({x: 1}); s
<detached primal solution from user>
>>> s.apply()
>>> x.value
1.0
```

If the solution was created by a solver (or attached to a problem via `attach_to`), more information is available:

```
>>> P = picos.Problem()
>>> P.minimize = x
>>> P += x >= 2
>>> s = P.solve(solver = "cvxopt", duals = False); s
<feasible primal solution (claimed optimal) from cvxopt>
>>> "{:.2f} ms".format(1000.0 * s.searchTime)
'0.83 ms'
>>> P += x >= 3; s
<infeasible primal solution (was feasible and claimed optimal) from cvxopt>
```

`__init__`(*primals*, *duals=None*, *problem=None*, *solver='user'*, *primalStatus=SS_UNKNOWN*, *dualStatus=SS_UNKNOWN*, *problemStatus=PS_UNKNOWN*, *searchTime=0.0*, *info=None*, *vectorizedPrimals=False*, *reportedValue=None*)

Create a solution to an optimization problem.

Parameters

- **primals** (*dict(picos.expressions.BaseVariable, object)*) – A mapping of variables to their primal solution value.
- **duals** (*dict(picos.constraints.Constraint, object)*) – A mapping of constraints to their dual solution value.
- **problem** (`picos.Problem`) – The problem that was solved to create the solution. If `None`, then the solution is “detached”.
- **solver** (*str*) – The name of the solver that was used to create the solution.
- **primalStatus** (*str*) – The primal solution status as reported by the solver.
- **dualStatus** (*str*) – The dual solution status as reported by the solver.
- **problemStatus** (*str*) – The state of the problem as reported by the solver.
- **searchTime** (*float*) – Seconds that the solution process took.
- **info** (*dict*) – Additional solution (meta)data.
- **vectorizedPrimals** (*bool*) – Whether primal solution values are given with respect to the variable’s special vectorization format as used by PICOS internally.
- **reportedValue** (*float*) – Objective value of the solution as reported by the solver.

`apply`(*primals=True*, *duals=True*, *clearOnNone=True*, *toProblem=None*, *snapshotStatus=False*)

Apply the solution to the involved variables and constraints.

Parameters

- **primals** (*bool*) – Whether to apply the primal solution.
- **duals** (*bool*) – Whether to apply the dual solution.
- **clearOnNone** (*bool*) – Whether to clear the value of a variable or constraint if the solution has it set to None. This could happen in case of an error or shortcoming of the solver or PICOS.

- **toProblem** (`picos.Problem`) – If set to a copy of the problem that was used to produce the solution, will apply the solution to that copy's variables and constraints instead.
- **snapshotStatus** (*bool*) – Whether to update the lastStatus attribute with the new (verified) solution status. PICOS enables this whenever it applies a solution returned by a solver.

attach_to(*problem*, *snapshotStatus=False*)

Attach (or move) the solution to a problem.

Only variables and constraints that exist on the problem (same name or ID, respectively) are kept.

Parameters
snapshotStatus (*bool*) – Whether to set the lastStatus attribute of the copy to match the new problem.

claimedStatus

The primal and dual solution status as claimed by the solver.

dualStatus

The dual solution status as claimed by the solver.

duals

The dual solution values returned by the solver.

info

Additional information provided by the solver.

lastStatus

The solution status as verified by PICOS when the solution was applied to the problem.

primalStatus

The primal solution status as claimed by the solver.

primals

The primal solution values returned by the solver.

problem

The problem that was solved to produce the solution.

problemStatus

The problem status as claimed by the solver.

reportedValue

The objective value of the solution as reported by the solver.

property reported_value

The objective value as reported by the solver, or `None`.

searchTime

Time in seconds that the solution search took.

solver

The solver that produced the solution.

property status

The current solution status as verified by PICOS.

> **Warning:** Accessing this attribute is expensive for large problems as a copy of the problem needs to be created and valued. If you have just applied the solution to a *`problem`*, query the solution's lastStatus attribute instead.

property value

The objective value of the solution as computed by PICOS.

> **Warning:** Accessing this attribute is expensive for large problems as a copy of the problem needs to be created and valued. If you have just applied the solution to a *problem*, query that problem instead.

vectorizedPrimals

Whether primal values refer to variables' special vectorizations.

Objects

picos.modeling.solution.**PS_FEASIBLE**

The problem is primal (and dual) feasible and bounded.

Default value

```
'feasible'
```

picos.modeling.solution.**PS_ILLPOSED**

The problem was found to be in a state that is not amenable to solution.

Default value

```
'illposed'
```

picos.modeling.solution.**PS_INFEASIBLE**

The problem is primal infeasible (and dual unbounded or infeasible).

Default value

```
'infeasible'
```

picos.modeling.solution.**PS_INF_OR_UNB**

The problem is primal infeasible or unbounded.

Being unbounded is usually infered from being dual infeasible.

Default value

```
'infeasible or unbounded'
```

picos.modeling.solution.**PS_UNBOUNDED**

The problem is primal unbounded (and dual infeasible).

Default value

```
'unbounded'
```

picos.modeling.solution.**PS_UNKNOWN**

The solver did not make a clear claim about the problem status.

Default value

```
'unknown'
```

`picos.modeling.solution.PS_UNSTABLE`

The problem was found numerically unstable or otherwise hard to handle.

Default value

```
'unstable'
```

`picos.modeling.solution.SS_EMPTY`

The solver claims not to have produced a solution.

Default value

```
'empty'
```

`picos.modeling.solution.SS_FAILURE`

The search was termined due to a solver failure.

Default value

```
'failure'
```

`picos.modeling.solution.SS_FEASIBLE`

The solution is feasible.

Default value

```
'feasible'
```

`picos.modeling.solution.SS_INFEASIBLE`

No feasible solution exists.

In the case of a primal solution, the problem is infeasible. In the case of a dual solution, the problem is unbounded.

Default value

```
'infeasible'
```

`picos.modeling.solution.SS_OPTIMAL`

The solution is optimal.

Default value

```
'optimal'
```

`picos.modeling.solution.SS_PREMATURE`

The search was prematurely terminated due to some limit.

Default value

```
'premature'
```

`picos.modeling.solution.SS_UNKNOWN`

The solver did not make a clear claim about the solution status.

Default value

```
'unknown'
```

`picos.modeling.solution.VS_DETACHED`

The solution is not attached to a problem (it was given by the user).

Default value

```
'detached'
```

picos.modeling.solution.**VS_DETACHED_EMPTY**

The solution is both detached and empty.

Default value

```
'detached empty'
```

picos.modeling.solution.**VS_EMPTY**

The solution is empty; there are neither primals nor duals.

Default value

```
'empty'
```

picos.modeling.solution.**VS_FEASIBLE**

The solution is primal feasible; there is no dual solution.

Default value

```
'feasible'
```

picos.modeling.solution.**VS_INCOMPLETE**

The primal (dual) solution does not concern all variables (constraints).

Default value

```
'incomplete'
```

picos.modeling.solution.**VS_INFEASIBLE**

The solution is primal infeasible; there is no dual solution.

Default value

```
'infeasible'
```

picos.modeling.solution.**VS_OUTDATED**

The solution does not fit the problem formulation any more.

Variables or constraints were removed from the problem.

Default value

```
'outdated'
```

picos.modeling.solution.**VS_PRIMAL_FEASIBLE**

The solution is primal feasible; a dual solution was not verified.

Default value

```
'primal feasible'
```

picos.modeling.solution.**VS_PRIMAL_INFEASIBLE**

The solution is primal infeasible; a dual solution was not verified.

Default value

```
'primal infeasible'
```

`picos.modeling.solution.VS_UNKNOWN`

PICOS failed to verify the solution.

Default value

```
'unverified'
```

picos.modeling.strategy

Optimization problem solution strategy search.

Exceptions

exception `picos.modeling.strategy.NoStrategyFound`

Bases: `RuntimeError`

No solution strategy found.

Raised when no viable combination of reformulations and solver to tackle the problem could be found.

Classes

class `picos.modeling.strategy.Strategy`(*problem*, *solver*, **reforms*)

Bases: `object`

Optimization problem solution strategy.

`__init__`(*problem*, *solver*, **reforms*)

Construct a *`Strategy`*.

Parameters

- **`problem`** (`Problem`) – The first step in the solution pipeline; the problem to be solved.
- **`solver`** (*`type`*) – The last step in the solution pipeline; the solver class to be used.
- **`reforms`** (*`list`*(`Reformulation`)) – Intermediate steps in the pipeline; reformulations to be applied. May not include *`ExtraOptions`* which is automatically made the first reformulation.

`execute`(***extra_options*)

Execute the solution strategy.

Parameters

`extra_options` – A keyword parameter sequence of additional options (in addition to those of the problem) to use for this search.

Returns

`Solution` to the problem.

classmethod `from_problem`(*problem*, ***extra_options*)

Create a solution strategy for the given problem.

Parameters

- **`problem`** (`Problem`) – The optimization problem to search a strategy for.
- **`extra_options`** – A keyword parameter sequence of additional options (in addition to those of the problem) to assume used.

`valid`(***extra_options*)

Whether the solution strategy can be executed.

Parameters

`extra_options` – A keyword parameter sequence of additional options (in addition to those of the problem) to assume used.

`property problem`

The problem to be solved.

`property reforms`

All reformulations in use.

This includes the implicit *`ExtraOptions`*.

`property solver`

The solver instance in use.

6.1.10 picos.reforms

Optimization problem reformulation recipes.

Classes

`class picos.reforms.`**`AbsoluteValueToAffineReformulation`**

See *`picos.reforms.reform_constraint.AbsoluteValueToAffineReformulation`*.

`class picos.reforms.`**`BallUncertainNormToRobustReformulation`**

See *`picos.reforms.reform_constraint.BallUncertainNormToRobustReformulation`*.

`class picos.reforms.`**`ComplexAffineToRealReformulation`**

See *`picos.reforms.reform_constraint.ComplexAffineToRealReformulation`*.

`class picos.reforms.`**`ComplexLMIToRealReformulation`**

See *`picos.reforms.reform_constraint.ComplexLMIToRealReformulation`*.

`class picos.reforms.`**`ConicQuadraticReformulation`**

See *`picos.reforms.reform_constraint.ConicQuadraticReformulation`*.

`class picos.reforms.`**`ConicallyUncertainAffineToRobustReformulation`**

See *`picos.reforms.reform_constraint.ConicallyUncertainAffineToRobustReformulation`*.

`class picos.reforms.`**`ConvexQuadraticToConicReformulation`**

See *`picos.reforms.reform_constraint.ConvexQuadraticToConicReformulation`*.

`class picos.reforms.`**`DetRootNReformulation`**

See *`picos.reforms.reform_constraint.DetRootNReformulation`*.

`class picos.reforms.`**`Dualization`**

See *`picos.reforms.reform_dualize.Dualization`*.

`class picos.reforms.`**`EpigraphReformulation`**

See *`picos.reforms.reform_epigraph.EpigraphReformulation`*.

`class picos.reforms.`**`ExtraOptions`**

See *`picos.reforms.reform_options.ExtraOptions`*.

`class picos.reforms.`**`ExtremumReformulation`**

See *`picos.reforms.reform_constraint.ExtremumReformulation`*.

class picos.reforms.**FlowReformulation**

See *picos.reforms.reform_constraint.FlowReformulation*.

class picos.reforms.**GeometricMeanToRSOCReformulation**

See *picos.reforms.reform_constraint.GeometricMeanToRSOCReformulation*.

class picos.reforms.**KullbackLeiblerToExpConeReformulation**

See *picos.reforms.reform_constraint.KullbackLeiblerToExpConeReformulation*.

class picos.reforms.**LogSumExpToExpConeReformulation**

See *picos.reforms.reform_constraint.LogSumExpToExpConeReformulation*.

class picos.reforms.**LogToExpConeReformulation**

See *picos.reforms.reform_constraint.LogToExpConeReformulation*.

class picos.reforms.**MatrixNormToVectorNormReformulation**

See *picos.reforms.reform_constraint.MatrixNormToVectorNormReformulation*.

class picos.reforms.**MomentAmbiguousExtremumAffineToDistributionallyRobustReformulation**

See *picos.reforms.reform_constraint.MomentAmbiguousExtremumAffineToDistributionallyRobustReformu*

class picos.reforms.**MomentAmbiguousSquaredNormToDistributionallyRobustReformulation**

See *picos.reforms.reform_constraint.MomentAmbiguousSquaredNormToDistributionallyRobustReformulat*

class picos.reforms.**NuclearNormReformulation**

See *picos.reforms.reform_constraint.NuclearNormReformulation*.

class picos.reforms.**PowerTraceReformulation**

See *picos.reforms.reform_constraint.PowerTraceReformulation*.

class picos.reforms.**ProductConeReformulation**

See *picos.reforms.reform_constraint.ProductConeReformulation*.

class picos.reforms.**Reformulation**

See *picos.reforms.reformulation.Reformulation*.

class picos.reforms.**ScenarioUncertainConicToRobustReformulation**

See *picos.reforms.reform_constraint.ScenarioUncertainConicToRobustReformulation*.

class picos.reforms.**SimplexToAffineReformulation**

See *picos.reforms.reform_constraint.SimplexToAffineReformulation*.

class picos.reforms.**SpectralNormReformulation**

See *picos.reforms.reform_constraint.SpectralNormReformulation*.

class picos.reforms.**SquaredNormToConicReformulation**

See *picos.reforms.reform_constraint.SquaredNormToConicReformulation*.

class picos.reforms.**SumExponentialsToConicReformulation**

See *picos.reforms.reform_constraint.SumExponentialsToConicReformulation*.

class picos.reforms.**SumExponentialsToLogSumExpReformulation**

See *picos.reforms.reform_constraint.SumExponentialsToLogSumExpReformulation*.

class picos.reforms.**SumExtremesReformulation**

See *picos.reforms.reform_constraint.SumExtremesReformulation*.

class picos.reforms.**VectorNormReformulation**

See *picos.reforms.reform_constraint.VectorNormReformulation*.

class
picos.reforms.**WassersteinAmbiguousExtremumAffineToDistributionallyRobustReformulation**

See *picos.reforms.reform_constraint.WassersteinAmbiguousExtremumAffineToDistributionallyRobustRe*

class
picos.reforms.**WassersteinAmbiguousSquaredNormToDistributionallyRobustReformulation**

See *picos.reforms.reform_constraint.WassersteinAmbiguousSquaredNormToDistributionallyRobustRefor*

class picos.reforms.**WeightedSumReformulation**

See *picos.reforms.reform_constraint.WeightedSumReformulation*.

Objects

picos.reforms.**SORTED_REFORMS**

A sequence of reformulations in topological order.

Default value

```
[<class 'picos.reforms.reform_epigraph.EpigraphReformulation'>,
 <class 'picos.reforms.reform_constraint.
↪AbsoluteValueToAffineReformulation'>,
 <class 'picos.reforms.reform_constraint.
↪BallUncertainNormToRobustReformulation'>,
 <class...
```

picos.reforms.reform_constraint

Reformulations that concern a particular type of constraint.

The reformulations' logic is not found here but defined within the constraint classes in the form of a *constraint conversion class*.

Classes

class picos.reforms.reform_constraint.**AbsoluteValueToAffineReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *AbsoluteValueConstraint.AffineConversion*.

CONSTRAINT_TYPE

alias of *AbsoluteValueConstraint*

CONVERSION_TYPE

alias of *AffineConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

`class picos.reforms.reform_constraint.`**`BallUncertainNormToRobustReformulation`**`(`*theObject*`)`

Bases: *`Reformulation`*

Reformulation created from *`BallUncertainNormConstraint.RobustConversion`*.

`CONSTRAINT_TYPE`

alias of *`BallUncertainNormConstraint`*

`CONVERSION_TYPE`

alias of *`RobustConversion`*

`__init__``(`*theObject*`)`

Implement *`__init__`*.

`backward``(`*solution*`)`

Implement *`backward`*.

`forward``()`

Implement *`forward`*.

`classmethod predict``(`*footprint*`)`

Implement *`predict`*.

`classmethod supports``(`*footprint*`)`

Implement *`supports`*.

`update``()`

Implement *`update`*.

`class picos.reforms.reform_constraint.`**`ComplexAffineToRealReformulation`**`(`*theObject*`)`

Bases: *`Reformulation`*

Reformulation created from *`ComplexAffineConstraint.RealConversion`*.

`CONSTRAINT_TYPE`

alias of *`ComplexAffineConstraint`*

`CONVERSION_TYPE`

alias of *`RealConversion`*

`__init__``(`*theObject*`)`

Implement *`__init__`*.

`backward``(`*solution*`)`

Implement *`backward`*.

`forward``()`

Implement *`forward`*.

`classmethod predict``(`*footprint*`)`

Implement *`predict`*.

`classmethod supports``(`*footprint*`)`

Implement *`supports`*.

`update``()`

Implement *`update`*.

`class picos.reforms.reform_constraint.`**`ComplexLMIToRealReformulation`**`(`*theObject*`)`

Bases: *`Reformulation`*

Reformulation created from *`ComplexLMIConstraint.RealConversion`*.

CONSTRAINT_TYPE
alias of *ComplexLMIConstraint*

CONVERSION_TYPE
alias of *RealConversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**ConicQuadraticReformulation**(*theObject*)
Bases: *Reformulation*

Reformulation created from *ConicQuadraticConstraint.Conversion*.

CONSTRAINT_TYPE
alias of *ConicQuadraticConstraint*

CONVERSION_TYPE
alias of *Conversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**ConicallyUncertainAffineToRobustReformulation**(*theObject*)
Bases: *Reformulation*

Reformulation created from *ConicallyUncertainAffineConstraint.RobustConversion*.

CONSTRAINT_TYPE
alias of *ConicallyUncertainAffineConstraint*

CONVERSION_TYPE
alias of *RobustConversion*

`__init__`(*theObject*)
Implement *`__init__`*.

`backward`(*solution*)
Implement *`backward`*.

`forward`()
Implement *`forward`*.

`classmethod predict`(*footprint*)
Implement *`predict`*.

`classmethod supports`(*footprint*)
Implement *`supports`*.

`update`()
Implement *`update`*.

`class picos.reforms.reform_constraint.`**`ConvexQuadraticToConicReformulation`**(*theObject*)
Bases: *`Reformulation`*

Reformulation created from *`ConvexQuadraticConstraint.ConicConversion`*.

`CONSTRAINT_TYPE`
alias of *`ConvexQuadraticConstraint`*

`CONVERSION_TYPE`
alias of *`ConicConversion`*

`__init__`(*theObject*)
Implement *`__init__`*.

`backward`(*solution*)
Implement *`backward`*.

`forward`()
Implement *`forward`*.

`classmethod predict`(*footprint*)
Implement *`predict`*.

`classmethod supports`(*footprint*)
Implement *`supports`*.

`update`()
Implement *`update`*.

`class picos.reforms.reform_constraint.`**`DetRootNReformulation`**(*theObject*)
Bases: *`Reformulation`*

Reformulation created from *`DetRootNConstraint.Conversion`*.

`CONSTRAINT_TYPE`
alias of *`DetRootNConstraint`*

`CONVERSION_TYPE`
alias of *`Conversion`*

`__init__`(*theObject*)
Implement *`__init__`*.

`backward`(*solution*)
Implement *`backward`*.

forward()

Implement `forward`.

classmethod **predict**(*footprint*)

Implement `predict`.

classmethod **supports**(*footprint*)

Implement `supports`.

update()

Implement `update`.

class picos.reforms.reform_constraint.**ExtremumReformulation**(*theObject*)

Bases: `Reformulation`

Reformulation created from `ExtremumConstraint.Conversion`.

CONSTRAINT_TYPE

alias of `ExtremumConstraint`

CONVERSION_TYPE

alias of `Conversion`

__init__(*theObject*)

Implement `__init__`.

backward(*solution*)

Implement `backward`.

forward()

Implement `forward`.

classmethod **predict**(*footprint*)

Implement `predict`.

classmethod **supports**(*footprint*)

Implement `supports`.

update()

Implement `update`.

class picos.reforms.reform_constraint.**FlowReformulation**(*theObject*)

Bases: `Reformulation`

Reformulation created from `FlowConstraint.Conversion`.

CONSTRAINT_TYPE

alias of `FlowConstraint`

CONVERSION_TYPE

alias of `Conversion`

__init__(*theObject*)

Implement `__init__`.

backward(*solution*)

Implement `backward`.

forward()

Implement `forward`.

classmethod **predict**(*footprint*)

Implement `predict`.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**GeometricMeanToRSOCReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *GeometricMeanConstraint.RSOCConversion*.

CONSTRAINT_TYPE

alias of *GeometricMeanConstraint*

CONVERSION_TYPE

alias of *RSOCConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**KullbackLeiblerToExpConeReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *KullbackLeiblerConstraint.ExpConeConversion*.

CONSTRAINT_TYPE

alias of *KullbackLeiblerConstraint*

CONVERSION_TYPE

alias of *ExpConeConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**LogSumExpToExpConeReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *LogSumExpConstraint.ExpConeConversion*.

CONSTRAINT_TYPE

alias of *LogSumExpConstraint*

CONVERSION_TYPE

alias of *ExpConeConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**LogToExpConeReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *LogConstraint.ExpConeConversion*.

CONSTRAINT_TYPE

alias of *LogConstraint*

CONVERSION_TYPE

alias of *ExpConeConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**MatrixNormToVectorNormReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *MatrixNormConstraint.VectorNormConversion*.

CONSTRAINT_TYPE
alias of *MatrixNormConstraint*

CONVERSION_TYPE
alias of *VectorNormConversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**MomentAmbiguousExtremumAffineToDistributionallyRobustReformula**

Bases: *Reformulation*

Reformulation created from *MomentAmbiguousExtremumAffineConstraint.DistributionallyRobustConversion*.

CONSTRAINT_TYPE
alias of *MomentAmbiguousExtremumAffineConstraint*

CONVERSION_TYPE
alias of *DistributionallyRobustConversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**MomentAmbiguousSquaredNormToDistributionallyRobustReformulatio**

Bases: *Reformulation*

Reformulation created from *MomentAmbiguousSquaredNormConstraint.DistributionallyRobustConversion*.

CONSTRAINT_TYPE
alias of *MomentAmbiguousSquaredNormConstraint*

CONVERSION_TYPE
alias of `DistributionallyRobustConversion`

__init__(*theObject*)
Implement `__init__`.

backward(*solution*)
Implement `backward`.

forward()
Implement `forward`.

classmethod predict(*footprint*)
Implement `predict`.

classmethod supports(*footprint*)
Implement `supports`.

update()
Implement `update`.

class picos.reforms.reform_constraint.**NuclearNormReformulation**(*theObject*)
Bases: `Reformulation`

Reformulation created from `NuclearNormConstraint.Conversion`.

CONSTRAINT_TYPE
alias of `NuclearNormConstraint`

CONVERSION_TYPE
alias of `Conversion`

__init__(*theObject*)
Implement `__init__`.

backward(*solution*)
Implement `backward`.

forward()
Implement `forward`.

classmethod predict(*footprint*)
Implement `predict`.

classmethod supports(*footprint*)
Implement `supports`.

update()
Implement `update`.

class picos.reforms.reform_constraint.**PowerTraceReformulation**(*theObject*)
Bases: `Reformulation`

Reformulation created from `PowerTraceConstraint.Conversion`.

CONSTRAINT_TYPE
alias of `PowerTraceConstraint`

CONVERSION_TYPE
alias of `Conversion`

__init__(*theObject*)
Implement `__init__`.

backward(*solution*)

Implement `backward`.

forward()

Implement `forward`.

classmethod predict(*footprint*)

Implement `predict`.

classmethod supports(*footprint*)

Implement `supports`.

update()

Implement `update`.

class picos.reforms.reform_constraint.**ProductConeReformulation**(*theObject*)

Bases: `Reformulation`

Reformulation created from `ProductConeConstraint.Conversion`.

CONSTRAINT_TYPE

alias of `ProductConeConstraint`

CONVERSION_TYPE

alias of `Conversion`

__init__(*theObject*)

Implement `__init__`.

backward(*solution*)

Implement `backward`.

forward()

Implement `forward`.

classmethod predict(*footprint*)

Implement `predict`.

classmethod supports(*footprint*)

Implement `supports`.

update()

Implement `update`.

class picos.reforms.reform_constraint.**ScenarioUncertainConicToRobustReformulation**(*theObject*)

Bases: `Reformulation`

Reformulation created from `ScenarioUncertainConicConstraint.RobustConversion`.

CONSTRAINT_TYPE

alias of `ScenarioUncertainConicConstraint`

CONVERSION_TYPE

alias of `RobustConversion`

__init__(*theObject*)

Implement `__init__`.

backward(*solution*)

Implement `backward`.

forward()

Implement `forward`.

classmethod predict(*footprint*)
: Implement *predict*.

classmethod supports(*footprint*)
: Implement *supports*.

update()
: Implement *update*.

class picos.reforms.reform_constraint.**SimplexToAffineReformulation**(*theObject*)
: Bases: *Reformulation*

  Reformulation created from *SimplexConstraint.AffineConversion*.

  CONSTRAINT_TYPE
  : alias of *SimplexConstraint*

  CONVERSION_TYPE
  : alias of *AffineConversion*

  __init__(*theObject*)
  : Implement *__init__*.

  backward(*solution*)
  : Implement *backward*.

  forward()
  : Implement *forward*.

  classmethod predict(*footprint*)
  : Implement *predict*.

  classmethod supports(*footprint*)
  : Implement *supports*.

  update()
  : Implement *update*.

class picos.reforms.reform_constraint.**SpectralNormReformulation**(*theObject*)
: Bases: *Reformulation*

  Reformulation created from *SpectralNormConstraint.Conversion*.

  CONSTRAINT_TYPE
  : alias of *SpectralNormConstraint*

  CONVERSION_TYPE
  : alias of *Conversion*

  __init__(*theObject*)
  : Implement *__init__*.

  backward(*solution*)
  : Implement *backward*.

  forward()
  : Implement *forward*.

  classmethod predict(*footprint*)
  : Implement *predict*.

  classmethod supports(*footprint*)
  : Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**SquaredNormToConicReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *SquaredNormConstraint.ConicConversion*.

CONSTRAINT_TYPE

alias of *SquaredNormConstraint*

CONVERSION_TYPE

alias of *ConicConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**SumExponentialsToConicReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *SumExponentialsConstraint.ConicConversion*.

CONSTRAINT_TYPE

alias of *SumExponentialsConstraint*

CONVERSION_TYPE

alias of *ConicConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**SumExponentialsToLogSumExpReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *SumExponentialsConstraint.LogSumExpConversion*.

CONSTRAINT_TYPE

alias of *SumExponentialsConstraint*

CONVERSION_TYPE

alias of *LogSumExpConversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**SumExtremesReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *SumExtremesConstraint.Conversion*.

CONSTRAINT_TYPE

alias of *SumExtremesConstraint*

CONVERSION_TYPE

alias of *Conversion*

__init__(*theObject*)

Implement *__init__*.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

class picos.reforms.reform_constraint.**VectorNormReformulation**(*theObject*)

Bases: *Reformulation*

Reformulation created from *VectorNormConstraint.Conversion*.

CONSTRAINT_TYPE
alias of *VectorNormConstraint*

CONVERSION_TYPE
alias of *Conversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**WassersteinAmbiguousExtremumAffineToDistributionallyRobustRefo**

Bases: *Reformulation*

Reformulation created from *WassersteinAmbiguousExtremumAffineConstraint.DistributionallyRobustConversion*.

CONSTRAINT_TYPE
alias of *WassersteinAmbiguousExtremumAffineConstraint*

CONVERSION_TYPE
alias of *DistributionallyRobustConversion*

__init__(*theObject*)
Implement *__init__*.

backward(*solution*)
Implement *backward*.

forward()
Implement *forward*.

classmethod predict(*footprint*)
Implement *predict*.

classmethod supports(*footprint*)
Implement *supports*.

update()
Implement *update*.

class picos.reforms.reform_constraint.**WassersteinAmbiguousSquaredNormToDistributionallyRobustReformu**

Bases: *Reformulation*

Reformulation created from *WassersteinAmbiguousSquaredNormConstraint.DistributionallyRobustConversion*.

CONSTRAINT_TYPE
alias of *WassersteinAmbiguousSquaredNormConstraint*

CONVERSION_TYPE
alias of `DistributionallyRobustConversion`

__init__(*theObject*)
Implement `__init__`.

backward(*solution*)
Implement `backward`.

forward()
Implement `forward`.

classmethod predict(*footprint*)
Implement `predict`.

classmethod supports(*footprint*)
Implement `supports`.

update()
Implement `update`.

class picos.reforms.reform_constraint.**WeightedSumReformulation**(*theObject*)
Bases: `Reformulation`

Reformulation created from `WeightedSumConstraint.Conversion`.

CONSTRAINT_TYPE
alias of `WeightedSumConstraint`

CONVERSION_TYPE
alias of `Conversion`

__init__(*theObject*)
Implement `__init__`.

backward(*solution*)
Implement `backward`.

forward()
Implement `forward`.

classmethod predict(*footprint*)
Implement `predict`.

classmethod supports(*footprint*)
Implement `supports`.

update()
Implement `update`.

picos.reforms.reform_dualize

Implementation of `Dualization`.

Classes

class picos.reforms.reform_dualize.**Dualization**(*theObject*)

Bases: *Reformulation*

Lagrange dual problem reformulation.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, LMIConstraint using RealVariable, ComplexVariable, SymmetricVariable, SkewSymmetricVariable, HermitianVariable, LowerTriangularVariable, UpperTriangularVariable and any options.>

picos.reforms.reform_epigraph

Implementation of *EpigraphReformulation*.

Classes

class picos.reforms.reform_epigraph.**EpigraphReformulation**(*theObject*)

Bases: *Reformulation*

Epigraph reformulation.

backward(*solution*)

Implement *backward*.

forward()

Implement *forward*.

classmethod predict(*footprint*)

Implement *predict*.

classmethod supports(*footprint*)

Implement *supports*.

update()

Implement *update*.

NEW_OBJECTIVE = <ExpressionType: AffineExpression[shape=(1, 1)|constant=False|nonneg=False]>

picos.reforms.reform_options

Implements a helper reformulation to apply temporary options.

Classes

`class picos.reforms.reform_options.`**`ExtraOptions`**`(`*theObject*`)`

Bases: `Reformulation`

Helper reformulation to apply temporary options.

This reformulation is different from all others in a number of ways:

- It doesn't change the footprint of any problem.
- It is automatically the first reformulation in any strategy.
- It is the only reformulation whose `execute` accepts a keyword argument sequence of additional options to use.
- It is the only reformulation that can be skipped entirely (by setting `self.output = self.input`).

The job of this reformulation is to apply temporary options passed to `solve` so that subsequent reformulations can find their options stored in their input problem.

`backward``(`*solution*`)`

Dummy-implement `backward`.

`execute``(`***extra_options*`)`

Override `execute`.

Adds the `extra_options` argument and attempts to perform as little reformulation work as possible.

`forward``()`

Dummy-implement `forward`.

`classmethod` **`predict`**`(`*footprint*`)`

Implement `predict`.

`classmethod` **`supports`**`(`*footprint*`)`

Implement `supports`.

`update``()`

Dummy-implement `update`.

picos.reforms.reformulation

Backend for problem reformulation classes.

Classes

`class picos.reforms.reformulation.`**`Reformulation`**`(`*theObject*`)`

Bases: `ABC`

Base class for problem reformulations.

Abstract base class for a reformulation from one (possibly already reformulated) problem form to another.

`__init__`(*theObject*)

Initialize *Reformulation* instances.

Parameters

theObject (Problem *or* Reformulation) – The input to work on; either an optimization problem or the (future) output of another reformulation.

`abstract backward`(*solution*)

Translate back a solution from reformulated to original problem.

Transforms a single *Solution* to *output* to a solution of *input*.

The method is allowed to modify the solution; it is not necessary to work on a copy. In particular, *attach_to* can be used if *forward* has created a deep copy of the problem.

`execute`()

Reformulate the problem and obtain a solution from the result.

For this to work there needs to be a solver instance at the end of the reformulation pipeline, which would implement its own version of this method that actually solves the problem and produces the first solution.

`abstract forward`()

Perform the initial problem reformulation.

Creates a modified copy or clone of the problem in *input* and stores it as *output*.

See *copy* and *clone* for the differences between a copy and a clone.

Implementations are supposed to do the necessary bookkeeping so that *backward* can transform a solution to the new problem back to a solution of the original problem.

`abstract classmethod predict`(*footprint*)

Predict the reformulation's effect on a problem footprint.

Given a problem footprint, returns another problem footprint that a problem with the former one would be reformulated to.

This is used to predict the effects of a reformulation when planning a solution strategy without the cost of actually transforming a problem.

`reset`()

Reset the pipeline from this reformulation onward.

This is done whenever a reformulation does not implement *update* so that succeeding reformulations do not attempt to update a problem which was completely rewritten as this may be inefficient.

`abstract classmethod supports`(*footprint*)

Whether the reformulation affects problems with the given footprint.

The reformulation must support every problem with such a footprint and the resulting problem should have a changed footprint.

`abstract update`()

Update a previous problem reformulation.

Updates *output* and related bookkeeping information with respect to changes in *input*.

Raises

NotImplementedError – If performing an update is not feasible for the reformulation.

`property input`

The input problem.

`output`

The output problem.

successor

The next reformulation in the pipeline.

property verbosity

Verbosity level of the reformulation; same as for input problem.

6.1.11 picos.settings

Contains global settings for PICOS.

All settings can be set via environment variables using the `PICOS_` prefix, e.g. `PICOS_SOLVER_WHITELIST='["cvxopt", "glpk"]' ./application.py` would set *`SOLVER_WHITELIST`* to `["cvxopt", "glpk"]` for this execution of `application.py` only. `Safe evaluation` is used to convert the given value to a Python object.

Applications that use PICOS may assign to these settings directly (silently overwriting any environment variable) but libraries that depend on PICOS should not do so as this would affect also applications and other libraries that use both PICOS and the library making the modificaiton.

Objects

`picos.settings.`**`ABSOLUTE_INTEGRALITY_TOLERANCE`**

Absolute tolerance used to validate integrality of integral variables.

Default value

```
0.0001
```

`picos.settings.`**`DEFAULT_CHARSET`**

Default charset to use for *`console output`*.

Can be any of `"ascii"`, `"latin1"` or `"unicode"` (default).

Note that applications can change the charset at any time using the respective function in the *`glyphs`* module.

Default value

```
'unicode'
```

`picos.settings.`**`INTERNAL_OPTIONS`**

Solution search options used whenever PICOS solves a problem internally.

By default, this limits the solver used to CVXOPT for reproducibility and to avoid licensing issues when non-free solvers are installed.

This setting is given as a dictionary. For keys and possible values see *`Options`*.

Default value

```
{'solver': 'cvxopt'}
```

`picos.settings.`**`LICENSE_WARNINGS`**

Let solvers ignore verbosity settings to print licensing related warnings.

License warnings are only printed if both *`LICENSE_WARNINGS`* and the solution search option *license_warnings* are set to true, or if the verbosity setting allows solver output in general.

Default value

```
True
```

`picos.settings.NONFREE_SOLVERS`

Whether PICOS may perform solution search with proprietary solvers.

Default value

```
True
```

`picos.settings.PREFER_GUROBI_MATRIX_INTERFACE`

Whether to prefer Gurobi's matrix interface when it is available.

This default can be overwritten by the *gurobi_matint* solver option.

Default value

```
True
```

`picos.settings.RELATIVE_HERMITIANNESS_TOLERANCE`

Relative tolerance used when checking whether a matrix is hermitian.

A matrix $A \in \mathbb{C}^{n \times n}$ is considered numerically hermitian if

$$\max_{1 \leq i,j \leq n} |(A - A^H)_{ij}| \leq \varepsilon \max_{1 \leq i,j \leq n} |A_{ij}|$$

where ε is this tolerance.

Default value

```
1e-10
```

`picos.settings.RELATIVE_SEMIDEFINITENESS_TOLERANCE`

Relative tolerance used when checking if a matrix is positive semidefinite.

A hermitian matrix $A \in \mathbb{C}^{n \times n}$ is considered numerically positive semidefinite if

$$\min \operatorname{eigvals}(A) \geq -\varepsilon \max\left(\{1\} \cup \operatorname{eigvals}(A)\right)$$

where ε is this tolerance.

Default value

```
1e-10
```

`picos.settings.RETURN_SOLUTION`

Whether `solve` returns a `Solution`.

Default value

```
True
```

`picos.settings.SOLVER_BLACKLIST`

A list of names of solvers that PICOS considers to be not available.

Default value

```
[]
```

`picos.settings.SOLVER_WHITELIST`

A list of names of solvers; PICOS considers all others not available.

Default value

```
[]
```

picos.settings.**UNRELIABLE_STRATEGIES**

Whether to pass solvers problems that they are known to struggle with.

This allows all problem/solver combinations that are skipped by "PICOS' choice".

Default value

```
False
```

picos.settings.**WARN_MISSING_DOCSTRINGS**

Whether to warn about missing docstrings for top level objects in a module.

Must be set via an environment variable to have an effect.

Default value

```
False
```

6.1.12 picos.solvers

Optimization solver interfaces.

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 name it to *solve* via the `solver` keyword argument.

Exceptions

exception picos.solvers.**ConflictingOptionsError**

See *picos.solvers.solver.ConflictingOptionsError*.

exception picos.solvers.**DependentOptionError**

See *picos.solvers.solver.DependentOptionError*.

exception picos.solvers.**OptionError**

See *picos.solvers.solver.OptionError*.

exception picos.solvers.**OptionValueError**

See *picos.solvers.solver.OptionValueError*.

exception picos.solvers.**ProblemUpdateError**

See *picos.solvers.solver.ProblemUpdateError*.

exception picos.solvers.**SolverError**

See *picos.solvers.solver.SolverError*.

exception picos.solvers.**UnsupportedOptionError**

See *picos.solvers.solver.UnsupportedOptionError*.

Classes

class picos.solvers.**CPLEXSolver**

See *picos.solvers.solver_cplex.CPLEXSolver*.

class picos.solvers.**CVXOPTSolver**

See *picos.solvers.solver_cvxopt.CVXOPTSolver*.

class picos.solvers.**ECOSSolver**

See *picos.solvers.solver_ecos.ECOSSolver*.

class picos.solvers.**GLPKSolver**

See *picos.solvers.solver_glpk.GLPKSolver*.

class picos.solvers.**GurobiSolver**

See *picos.solvers.solver_gurobi.GurobiSolver*.

class picos.solvers.**MOSEKFusionSolver**

See *picos.solvers.solver_mskfsn.MOSEKFusionSolver*.

class picos.solvers.**MOSEKSolver**

See *picos.solvers.solver_mosek.MOSEKSolver*.

class picos.solvers.**OSQPSolver**

See *picos.solvers.solver_osqp.OSQPSolver*.

class picos.solvers.**QICSSolver**

See *picos.solvers.solver_qics.QICSSolver*.

class picos.solvers.**SCIPSolver**

See *picos.solvers.solver_scip.SCIPSolver*.

class picos.solvers.**SMCPSolver**

See *picos.solvers.solver_smcp.SMCPSolver*.

class picos.solvers.**Solver**

See *picos.solvers.solver.Solver*.

Functions

picos.solvers.**all_solvers**()

Return a dictionary mapping solver names to implementation classes.

picos.solvers.**available_solvers**(*problem=None*)

Return a sorted list of names of available solvers.

Parameters
problem – DEPRECATED

picos.solvers.**get_solver**(*name*)

Return the implementation class of the solver with the given name.

picos.solvers.**get_solver_name**(*solver*)

Return the registry name of a solver instance.

picos.solvers.solver

Backend for solver interface implementations.

Exceptions

exception picos.solvers.solver.**ConflictingOptionsError**

Bases: *OptionError*

Two solver options are in conflict.

Raised by implementations of `_solve` to signal to the user that two options they specified cannot be used in conjunction.

exception `picos.solvers.solver.`**`DependentOptionError`**

Bases: *`OptionError`*

A solver option is invalid due to another option not being set.

Raised by implementations of `_solve` to signal to the user that an option they specified needs another option to also be set.

exception `picos.solvers.solver.`**`OptionError`**

Bases: *`SolverError`*

Base class for solver option related errors.

exception `picos.solvers.solver.`**`OptionValueError`**

Bases: *`OptionError`*, `ValueError`

A solver option has an invalid value.

Raised by implementations of `_solve` to signal to the user that they have set an option to an invalid value.

exception `picos.solvers.solver.`**`ProblemUpdateError`**

Bases: *`SolverError`*

Changes to the problem could not be forward to the solver.

Raised by implementations of `_update_problem` to signal to the method `_load_problem` that the problem needs to be re-imported.

exception `picos.solvers.solver.`**`SolverError`**

Bases: `Exception`

Base class for solver-specific exceptions.

exception `picos.solvers.solver.`**`UnsupportedOptionError`**

Bases: *`OptionError`*

The solver does not support an option.

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.

Classes

class `picos.solvers.solver.`**`Solver`**(*problem*)

Bases: `ABC`

Base class for an interface to an optimization solver.

`__init__`(*problem*)

Instanciate a solver interface with an optimization problem.

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 a solution is requested.

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.

classmethod **`available`**(*verbose=False*)

Whether the solver is properly installed on the system.

`static check_import(importName)`

Asserts that a module is available without actually importing it.

Raises
`ModuleNotFoundError` – If the module could not be found.

`abstract classmethod default_penalty()`

Report the default penalty for the solver.

See *`Options`* for the scale.

`execute()`

Solve the problem and return the solution.

Returns picos.Solution or list(picos.Solution)
A solution object or list thereof.

`external_problem()`

Return the external (PICOS) problem represenation.

`classmethod get_via_name(interface_in_parenthesis=False)`

Return the name of the solver with the Python interface used.

`internal_problem()`

Return the solver's internal problem represenation.

`abstract classmethod is_free()`

Report whether the solver is free software.

This allows users to prevent PICOS from using non-free solvers at all, including for internal use, via the *`NONFREE_SOLVERS`* setting.

`abstract classmethod names()`

Return a name sequence `(internal, short, long, interface)`.

1. The internal name is a lowercase keyword used for solver selection.
2. The short name is a properly capitalized official solver shortand.
3. The long name is the full official name of the solver.
4. The interface name is a properly capitalized short name of the Python interface used, or `None` if the solver is Python-native or includes a unique Python interface.

`classmethod penalty(options)`

Report solver penalty given an *`Options`* object.

`classmethod predict(footprint)`

Return the solver class.

This mimics the behavior of *`Reformulation.predict`* so that solvers can be the last pipeline node in a reformulation strategy.

`reset()`

A shorthand for *`reset_problem`*.

This is defined for consistency with *`Reformulation.reset`*.

`abstract reset_problem()`

Reset 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.

abstract classmethod supports(*footprint*, *explain=False*)

Whether a type of problem, given by footprint, is supported.

The default implementation ensures that all reformulations required by user's choice have been performed before the problem is handed to the solver. Solver implementations are thus required to incorporate it.

Parameters

explain (*bool*) – If `True`, then this returns a `tuple` where the first element is this method's regular return value and where the second element is a string naming one reason why the footprint is not supported (`None` if it is).

abstract classmethod test_availability()

Raise an exception if the solver is not installed on the system.

Checks whether the solver is installed on the system, and raises an appropriate exception (usually `ModuleNotFoundError` or `ImportError`) if not. Does not return anything.

verbosity()

Return the problem's current verbosity level.

property ext

The "external" (input) problem.

property interface_name

Short name of the Python interface used, or `None`.

property long_name

Long name of the solver.

property name

Keyword string of the solver.

property short_name

Short name of the solver.

property via_name

The short names of the solver and Python interface used.

picos.solvers.solver_cplex

Implementation of *CPLEXSolver*.

Classes

class picos.solvers.solver_cplex.**CPLEXSolver**(*problem*)

Bases: *Solver*

Interface to the CPLEX solver via its official Python interface.

Note: Names are used instead of indices for identifying both variables and constraints since indices can change if the CPLEX instance is modified.

class MetaConstraint(*con*, *dim*)

Bases: `tuple`

static __new__(*_cls*, *con*, *dim*)

Create new instance of MetaConstraint(con, dim)

con

Alias for field number 0

dim

Alias for field number 1

__init__(*problem*)

Initialize a CPLEX solver interface.

Parameters

problem (`Problem`) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

NONCONVEX_QP = <Specification: Optimize QuadraticExpression subject to DummyConstraint, AffineConstraint using any variables and any options.>

SUPPORTED = <Specification: Optimize AffineExpression, QuadraticExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, ConvexQuadraticConstraint using any variables and any options.>

Objects

picos.solvers.solver_cplex.**CPLEX_STATUS_CODES**

Maps CPLEX status code to PICOS status triples.

Default value

```
{1: ('optimal', 'optimal', 'feasible'),
 2: ('unknown', 'infeasible', 'unbounded'),
 3: ('infeasible', 'unknown', 'infeasible'),
 4: ('unknown', 'unknown', 'infeasible or unbounded'),
 5: ('infeasible', 'unknown', 'unstable'),
 6: ('unknown',...
```

picos.solvers.solver_cvxopt

Implementation of *CVXOPTSolver*.

Classes

class picos.solvers.solver_cvxopt.**CVXOPTSolver**(*problem*)

Bases: *Solver*

Interface to the CVXOPT solver.

Also used as an interface to the *SMCP solver*.

__init__(*problem*)

Initialize a CVXOPT solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression, LogSumExp subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, LMIConstraint, LogSumExpConstraint using RealVariable, ComplexVariable, SymmetricVariable, SkewSymmetricVariable, HermitianVariable, LowerTriangularVariable, UpperTriangularVariable and any options.>

property is_smcp

Whether to implement SMCP instead of CVXOPT.

picos.solvers.solver_ecos

Implementation of *ECOSSolver*.

Classes

class picos.solvers.solver_ecos.**ECOSSolver**(*problem*)

Bases: *Solver*

Interface to the ECOS solver via its official Python interface.

__init__(*problem*)

Initialize an ECOS solver interface.

Parameters
problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

static stack(**args*)

Stack vectors or matrices, the latter vertically.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, ExpConeConstraint using any variables and any options.>

property ecos

Return the ECOS core module ecos.py.

The module is obtained by `import ecos` up to ECOS 2.0.6 and by `import ecos.ecos` starting with ECOS 2.0.7.

picos.solvers.solver_glpk

Implementation of *GLPKSolver*.

Classes

class picos.solvers.solver_glpk.**GLPKSolver**(*problem*)

Bases: *Solver*

Interface to the GLPK solver via swiglpk.

__init__(*problem*)

Initialize a GLPK solver interface.

Parameters
problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint using any variables and any options.>

UNUSED_VAR = UnusedVar(dim=1)

picos.solvers.solver_gurobi

Implementation of *GurobiSolver*.

Classes

class picos.solvers.solver_gurobi.**GurobiSolver**(*problem*)

Bases: *Solver*

Interface to the Gurobi solver via its official Python interface.

class GurobiMetaConstraint(*auxCons*, *auxVars*)

Bases: tuple

static __new__(*_cls*, *auxCons*, *auxVars*)

Create new instance of GurobiMetaConstraint(auxCons, auxVars)

auxCons

Alias for field number 0

auxVars

Alias for field number 1

__init__(*problem*)

Initialize a Gurobi solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod **supports**(*footprint*, *explain=False*)

Implement *supports*.

classmethod **test_availability**()

Implement *test_availability*.

SUPPORTED_8 = <Specification: Optimize AffineExpression, QuadraticExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, ConvexQuadraticConstraint using any variables and any options.>

SUPPORTED_9 = <Specification: Optimize AffineExpression, QuadraticExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, NonconvexQuadraticConstraint using any variables and any options.>

property **matint**

Whether Gurobi's matrix interface is in use.

picos.solvers.solver_mosek

Implementation of *MOSEKSolver*.

Classes

class picos.solvers.solver_mosek.**MOSEKSolver**(*problem*)

Bases: *Solver*

Interface to the MOSEK solver via its low level Optimizer API.

Supports both MOSEK 8 and 9.

The low level API is tedious to interface, but is currently much faster than the high level Fusion API, which would be the prefered interface otherwise.

__init__(*problem*)

Initialize a MOSEK (Optimizer) solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod **default_penalty**()

Implement *default_penalty*.

classmethod **is_free**()

Implement *is_free*.

classmethod **names**()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod **supports**(*footprint*, *explain=False*)

Implement *supports*.

classmethod **test_availability**()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression, QuadraticExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, ConvexQuadraticConstraint, LMIConstraint using any variables and any options.>

property 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.")

property ver

The major version of the available MOSEK library.

picos.solvers.solver_mskfsn

Implementation of *MOSEKFusionSolver*.

Classes

class picos.solvers.solver_mskfsn.**MOSEKFusionSolver**(*problem*)

Bases: *Solver*

Interface to the MOSEK solver via its high level Fusion API.

Supports both MOSEK 8 and 9.

The Fusion API is currently much slower than MOSEK's low level Python API. If this changes in the future, the Fusion API would be the prefered interface.

__init__(*problem*)

Initialize a MOSEK (Fusion) solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, LMIConstraint using any variables and any options.>

property ver

The major version of the available MOSEK library.

picos.solvers.solver_osqp

Implementation of *OSQPSolver*.

Classes

class picos.solvers.solver_osqp.**OSQPSolver**(*problem*)

Bases: *Solver*

Interface to the OSQP solver.

__init__(*problem*)

Initialize a OSQP solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression, QuadraticExpression subject to DummyConstraint, AffineConstraint using RealVariable, ComplexVariable, SymmetricVariable, SkewSymmetricVariable, HermitianVariable, LowerTriangularVariable, UpperTriangularVariable and any options.>

picos.solvers.solver_qics

Implementation of *QICSSolver*.

Classes

class picos.solvers.solver_qics.**QICSSolver**(*problem*)

Bases: *Solver*

Interface to the QICS solver.

__init__(*problem*)

Initialize a QICS solver interface.

Parameters

problem (Problem) – The problem to be solved.

static blkdiag(**args*)

Stack vectors or matrices.

static complex_to_real(*A*)

Splits complex matrix into real and imaginary parts.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

static stack(**args*)

Stack vectors or matrices.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, LMIConstraint, ComplexLMIConstraint, ExpConeConstraint, KullbackLeiblerConstraint, QuantRelEntropyConstraint, ComplexQuantRelEntropyConstraint, QuantCondEntropyConstraint, ComplexQuantCondEntropyConstraint, QuantKeyDistributionConstraint, ComplexQuantKeyDistributionConstraint, OpRelEntropyConstraint, ComplexOpRelEntropyConstraint, TrOpRelEntropyConstraint, ComplexTrOpRelEntropyConstraint, MatrixGeoMeanEpiConstraint, ComplexMatrixGeoMeanEpiConstraint, TrMatrixGeoMeanEpiConstraint, ComplexTrMatrixGeoMeanEpiConstraint, MatrixGeoMeanHypoConstraint, ComplexMatrixGeoMeanHypoConstraint, TrMatrixGeoMeanHypoConstraint, ComplexTrMatrixGeoMeanHypoConstraint using RealVariable, ComplexVariable, SymmetricVariable, SkewSymmetricVariable, HermitianVariable, LowerTriangularVariable, UpperTriangularVariable and any options.>

picos.solvers.solver_scip

Implementation of *SCIPSolver*.

Classes

class picos.solvers.solver_scip.**SCIPSolver**(*problem*)

Bases: *Solver*

Interface to the SCIP solver via the PySCIPOpt Python interface.

__init__(*problem*)

Initialize a SCIP solver interface.

Parameters

problem (Problem) – The problem to be solved.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

reset_problem()

Implement *reset_problem*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, SOCConstraint, RSOCConstraint, ConvexQuadraticConstraint, NonconvexQuadraticConstraint using any variables and any options.>

picos.solvers.solver_smcp

Implementation of *SMCPSolver*.

Classes

class picos.solvers.solver_smcp.**SMCPSolver**(*problem*)

Bases: *CVXOPTSolver*

Interface to the SMCP solver.

Most of the logic is implemented in the *CVXOPTSolver* base class.

classmethod default_penalty()

Implement *default_penalty*.

classmethod is_free()

Implement *is_free*.

classmethod names()

Implement *names*.

classmethod supports(*footprint*, *explain=False*)

Implement *supports*.

classmethod test_availability()

Implement *test_availability*.

SUPPORTED = <Specification: Optimize AffineExpression subject to DummyConstraint, AffineConstraint, LMIConstraint using RealVariable, ComplexVariable, SymmetricVariable, SkewSymmetricVariable, HermitianVariable, LowerTriangularVariable, UpperTriangularVariable and any options.>

property is_smcp

Whether to implement SMCP instead of CVXOPT.

6.1.13 picos.tools

Backwards compatibility version of an older module.

Functions

picos.tools.**ball**()

See *picos.expressions.algebra.ball*.

picos.tools.**detect_range**()

See *picos.formatting.detect_range*.

picos.tools.**detrootn**()

See *picos.expressions.algebra.detrootn*.

picos.tools.**diag**()

See *picos.expressions.algebra.diag*.

picos.tools.**diag_vect**()

See *picos.expressions.algebra.diag_vect*.

picos.tools.**eval_dict**(*dict_of_variables*)

Evaluate all values of a dictionary.

Parameters
dict_of_variables (*dict*) – A dictionary mapping arbitrary keys to PICOS objects that have a value attribute, such as variables.

Returns
Another dictionary with the original dicitonary's values replaced by the value of their value attribute.

Deprecated since version 2.0: Write '{k: v.value for k, v in x.items()}' instead.

picos.tools.**exp**()

See *picos.expressions.algebra.exp*.

picos.tools.**expcone**()

See *picos.expressions.algebra.expcone*.

picos.tools.**flow_Constraint**()

See *picos.expressions.algebra.flow_Constraint*.

picos.tools.**geomean**()

See *picos.expressions.algebra.geomean*.

picos.tools.**import_cbf**()

See *picos.modeling.file_in.import_cbf*.

picos.tools.**kron**()

See *picos.expressions.algebra.kron*.

picos.tools.**kullback_leibler**()

See *picos.expressions.algebra.kullback_leibler*.

picos.tools.**lambda_max**()

See *picos.expressions.algebra.lambda_max*.

picos.tools.**lambda_min**()

See *picos.expressions.algebra.lambda_min*.

`picos.tools.log()`

See *picos.expressions.algebra.log*.

`picos.tools.logsumexp()`

See *picos.expressions.algebra.logsumexp*.

`picos.tools.lse()`

See *picos.expressions.algebra.lse*.

`picos.tools.new_param()`

See *picos.expressions.algebra.new_param*.

`picos.tools.norm()`

See *picos.expressions.algebra.norm*.

`picos.tools.parameterized_string()`

See *picos.formatting.parameterized_string*.

`picos.tools.partial_trace()`

See *picos.expressions.algebra.partial_trace*.

`picos.tools.partial_transpose()`

See *picos.expressions.algebra.partial_transpose*.

`picos.tools.retrieve_matrix`(*mat*, *exSize=None*)

Legacy wrapper around `load_data`.

Deprecated since version 2.0: Use `picos.expressions.data.load_data` instead.

`picos.tools.simplex()`

See *picos.expressions.algebra.simplex*.

`picos.tools.sum()`

See *picos.expressions.algebra.sum*.

`picos.tools.sum_k_largest()`

See *picos.expressions.algebra.sum_k_largest*.

`picos.tools.sum_k_largest_lambda()`

See *picos.expressions.algebra.sum_k_largest_lambda*.

`picos.tools.sum_k_smallest()`

See *picos.expressions.algebra.sum_k_smallest*.

`picos.tools.sum_k_smallest_lambda()`

See *picos.expressions.algebra.sum_k_smallest_lambda*.

`picos.tools.sumexp()`

See *picos.expressions.algebra.sumexp*.

`picos.tools.trace()`

See *picos.expressions.algebra.trace*.

`picos.tools.tracepow()`

See *picos.expressions.algebra.tracepow*.

`picos.tools.truncated_simplex()`

See *picos.expressions.algebra.truncated_simplex*.

Objects

`picos.tools.builtin_sum`

Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

Default value

```
<built-in function sum>
```

6.1.14 picos.uncertain

Tools for handling uncertain data.

Exceptions

exception `picos.uncertain.IntractableWorstCase`

See *picos.expressions.uncertain.uexpression.IntractableWorstCase*.

Classes

class `picos.uncertain.ConicPerturbationSet`

See *picos.expressions.uncertain.pert_conic.ConicPerturbationSet*.

class `picos.uncertain.MomentAmbiguitySet`

See *picos.expressions.uncertain.pert_moment.MomentAmbiguitySet*.

class `picos.uncertain.ScenarioPerturbationSet`

See *picos.expressions.uncertain.pert_scenario.ScenarioPerturbationSet*.

class `picos.uncertain.UnitBallPerturbationSet`

See *picos.expressions.uncertain.pert_conic.UnitBallPerturbationSet*.

class `picos.uncertain.WassersteinAmbiguitySet`

See *picos.expressions.uncertain.pert_wasserstein.WassersteinAmbiguitySet*.

6.1.15 picos.valuable

Common interface for objects that can have a numeric value.

Exceptions

exception `picos.valuable.NotValued`

Bases: `RuntimeError`

The operation cannot be performed due to a mutable without a value.

Note that the *value*, *value_as_matrix*, *np*, and *np2d* attributes do not raise this exception, but return `None` instead.

Classes

class picos.valuable.**Valuable**

Bases: ABC

Abstract base class for objects that can have a numeric value.

This is used by all algebraic expressions through their *Expression* base class as well as by *Objective* and, referencing the latter, by *Problem* instances.

__array__(*dtype=None*)

Return the value as a NumPy array.

__complex__()

Cast the value to a complex.

__float__()

Cast the value to a float.

__index__()

Propose the value as an index.

__int__()

Cast the value to an int.

__round__(*ndigits=None*)

Round the value to a certain precision.

property np

Value of the object as a NumPy type, or None.

Refer to *value* for when it is defined (not None).

Returns

A one- or two-dimensional numpy.ndarray, if the value is a vector or a matrix, respectively, or a NumPy scalar type such as numpy.float64, if the value is a scalar, or None, if the value is not defined.

Distinction

- Like *value* and *np2d*, an undefined value is returned as None.
- Unlike *value*, scalars are returned as NumPy scalar types as opposed to Python builtin scalar types while vectors and matrices are returned as NumPy arrays as opposed to CVXOPT matrices.
- Unlike *np2d*, scalars are returned as NumPy scalar types and vectors are returned as NumPy one-dimensional arrays as opposed to always returning two-dimensional arrays.

Example

```
>>> from picos import ComplexVariable
>>> Z = ComplexVariable("Z", (3, 3))
>>> Z.value = [i + i*1j for i in range(9)]
```

Proper matrices are return as 2D arrays:

```
>>> Z.value  # CVXOPT matrix.
<3x3 matrix, tc='z'>
>>> Z.np  # NumPy 2D array.
array([[0.+0.j, 3.+3.j, 6.+6.j],
       [1.+1.j, 4.+4.j, 7.+7.j],
       [2.+2.j, 5.+5.j, 8.+8.j]])
```

Both row and column vectors are returned as 1D arrays:

```
>>> z = Z[:,0]  # First column of Z.
>>> z.value.size  # CVXOPT column vector.
(3, 1)
>>> z.T.value.size  # CVXOPT row vector.
(1, 3)
>>> z.value == z.T.value
False
>>> z.np.shape  # NumPy 1D array.
(3,)
>>> z.T.np.shape  # Same array.
(3,)
>>> from numpy import array_equal
>>> array_equal(z.np, z.T.np)
True
```

Scalars are returned as NumPy types:

```
>>> u = Z[0,0]  # First element of Z.
>>> type(u.value)  # Python scalar.
<class 'complex'>
>>> type(u.np)  # NumPy scalar.
<class 'numpy.complex128'>
```

Undefined values are returned as None:

```
>>> del Z.value
>>> Z.value is Z.np is None
True
```

property np2d

Value of the object as a 2D NumPy array, or `None`.

Refer to *value* for when it is defined (not `None`).

Returns
The value as a two-dimensional `numpy.ndarray`, or `None`, if the value is not defined.

Distinction

- Like *np*, values are returned as NumPy types or `None`.
- Unlike *np*, both scalar and vectorial values are returned as two-dimensional arrays. In particular, row and column vectors are distinguished.

property safe_value

Value of the object, if defined.

Refer to *value* for when it is defined.

Returns
The value as a Python scalar or CVXOPT matrix.

Raises
NotValued – If the value is not defined.

Distinction

- Unlike *value*, an undefined value raises an exception.
- Like *value*, scalars are returned as scalar types.

property safe_value_as_matrix

Value of the object as a CVXOPT matrix type, if defined.

Refer to *value* for when it is defined.

Returns
The value as a CVXOPT matrix.

Raises
NotValued – If the value is not defined.

Distinction

- Unlike *value*, an undefined value raises an exception.
- Unlike *value*, scalars are returned as 1×1 matrices.

property sp

Value as a ScipPy sparse matrix or a NumPy 2D array or `None`.

If PICOS stores the value internally as a CVXOPT sparse matrix, or equivalently if *value_as_matrix* returns an instance of `cvxopt.spmatrix`, then this returns the value as a `SciPy sparse matrix in CSC format`. Otherwise, this property is equivalent to *np2d* and returns a two-dimensional NumPy array, or `None`, if the value is undefined.

Example

```
>>> import picos, cvxopt
>>> X = picos.RealVariable("X", (3, 3))
>>> X.value = cvxopt.spdiag([1, 2, 3])  # Stored as a sparse matrix.
>>> type(X.value)
<class 'cvxopt.base.spmatrix'>
>>> type(X.sp)
<class 'scipy.sparse._csc.csc_matrix'>
>>> X.value = range(9)  # Stored as a dense matrix.
>>> type(X.value)
<class 'cvxopt.base.matrix'>
>>> type(X.sp)
<class 'numpy.ndarray'>
```

property value

Value of the object, or `None`.

For an expression, it is defined if the expression is constant or if all mutables involved in the expression are valued. Mutables can be valued directly by writing to their *value* attribute. Variables are also valued by PICOS when an optimization solution is found.

Some expressions can also be valued directly if PICOS can find a minimal norm mutable assignment that makes the expression have the desired value. In particular, this works with affine expressions whose linear part has an under- or well-determined coefficient matrix.

If you prefer the value as a NumPy, use *np* instead.

Returns
The value as a Python scalar or CVXOPT matrix, or `None` if it is not defined.

Distinction

- Unlike *safe_value* and *safe_value_as_matrix*, an undefined value is returned as `None`.
- Unlike *value_as_matrix* and *safe_value_as_matrix*, scalars are returned as scalar types.
- For uncertain expressions, see also *worst_case_value*.

Example

```
>>> from picos import RealVariable
>>> x = RealVariable("x", (1,3))
>>> y = RealVariable("y", (1,3))
>>> e = x - 2*y + 3
>>> print("e:", e)
e: x - 2·y + [3]
>>> e.value = [4, 5, 6]
>>> print("e: ", e, "\nx: ", x, "\ny: ", y, sep = "")
e: [ 4.00e+00  5.00e+00  6.00e+00]
x: [ 2.00e-01  4.00e-01  6.00e-01]
y: [-4.00e-01 -8.00e-01 -1.20e+00]
```

property value_as_matrix

Value of the object as a CVXOPT matrix type, or None.

Refer to *value* for when it is defined (not None).

Returns
The value as a CVXOPT matrix, or None if it is not defined.

Distinction

- Like *value*, an undefined value is returned as None.
- Unlike *value*, scalars are returned as 1×1 matrices.

property valued

Whether the object is valued.

Note: Querying this attribute is *not* faster than immediately querying *value* and checking whether it is None. Use it only if you do not need to know the value, but only whether it is available.

Example

```
>>> from picos import RealVariable
>>> x = RealVariable("x", 3)
>>> x.valued
False
>>> x.value
>>> print(x.sum)
∑(x)
>>> x.value = [1, 2, 3]
>>> x.sum.valued
True
>>> print(x.sum)
6.0
```

Functions

`picos.valuable.`**`patch_scipy_array_priority`**`()`

Monkey-patch scipy.sparse to make it respect `__array_priority__`.

This works around https://github.com/scipy/scipy/issues/4819 and is inspired by CVXPY's scipy_wrapper.py.

CHAPTER

SEVEN

CHANGELOG

This file documents major changes to PICOS. The format is based on Keep a Changelog.

7.1 2.5 - 2024-10-20

The quantum relative entropy update.

Important

- The `complex inner product` now follows the physics convention of conjugate linearity in the first and linearity in the second argument. Thus `(A | B)` now represents $\langle A \mid B\rangle = \mathrm{Tr}(A^\dagger B)$ instead of $\mathrm{Tr}(B^\dagger A)$. The change amounts to taking the complex conjugate of the result and only affects complex operands which are not both Hermitian.

Added

- Support for QICS, a new conic solver which supports solving LP, SOCP, SDP, EXP, as well as a new class of problems called quantum relative entropy programs (QREP).
- New expressions for quantum entropic and nonlinear matrix-valued functions, including arising in quantum information theory, as well as constraints representing the epigraphs or hypographs of these functions. These include
  - Quantum entropy (`picos.quantentr`),
  - Quantum relative entropy (`picos.quantrelentr`),
  - Quantum conditional entropy (`picos.quantcondentr`),
  - Operator relative entropy (`picos.oprelentr`), and
  - Matrix geometric mean (`picos.mtxgeomean`).
- Documentation examples for solving quantum relative entropy programs.
- Test cases for new quantum entropy and non-commutative perspective functions.
- Affine expression properties `opreal` (equals `hermitianized`) and `opimag`.

Changed

- Kronecker products of sparse matrices are now computed using SciPy, if available.
- Updated *KullbackLeiblerConstraint* so that it can be used as a constraint without having to convert to the exponential cone.
- CI/CD now uses ruff instead of pylama for linting.

Fixed

- Indexing into a Gurobi variable container which cannot be indexed anymore.
- Creating a squared Frobenius norm of a complex expression.
- *WeightedSum* not working for a single weighted expression.
- A doctest failure related to *patch_scipy_array_priority*.
- A doctest failure related to *add_variable*.

Deprecated

- The `rng` argument in *shuffled*, which is no longer a parameter in `random.shuffle` as of Python 3.11.

7.2 2.4 - 2022-02-12

The performance update.

Added

- Support for noncovex quadratic constraints with Gurobi 9 (or later).
- Setting *UNRELIABLE_STRATEGIES* to enable passing of problems to solvers that nominally support them but have proven unreliable.
- Setting *PREFER_GUROBI_MATRIX_INTERFACE* and option *gurobi_matint* to toggle between Gurobi's legacy and matrix interface.
- Option *mosek_basic_sol* to let MOSEK (Optimizer) compute a basic solution for LPs.

Changed

- The performance for solving problems with large data has been improved
  - drastically for CVXOPT and MOSEK (Optimizer; LPs in particular),
  - significantly for Cplex and SCIP, and
  - subtly for GLPK, Gurobi and ECOS.

  This is most notable for LPs with a dense constraint matrix where the overhead for data passing can be significant in relation to the search time.
- The performance of *picos.sum* when summing a large number of (bi-)affine expressions has been improved drastically.
- When possible, Gurobi is now interfaced through its matrix interface, which is faster for large data. This requires Gurobi 9 (or later) and SciPy.
- By default, solving with MOSEK (Optimizer) does not return a basic LP solution any more. Use *mosek_basic_sol* to control this.

- The default value of *cvxopt_kktsolver* is now `None` and means "try the fast `"chol"` first and fall back to the reliable `"ldl"` on error".
- Dualization now makes use of variable bounds to reduce the number of auxiliary constraints.
- The Python interface used to communicate with a solver is now mentioned in various log messages and exceptions.

Fixed

- On-the-fly loading of a data vector in a multiplication with a matrix expression.
- Maximization of a squared norm not being detected as a nonconvex quadratic objective and being passed to solvers that do not support it.

7.3 2.3 - 2021-10-07

The syntactic sugar update.

Important

- When forming linear matrix inequalities with the << or >> operator, if one operand is an $n \times n$ matrix and the other is an n-dimensional vector (or a scalar), the latter is now understood as (respectively broadcasted along) the main diagonal of an $n \times n$ diagonal matrix. In particular `X >> 1` is now understood as $X \succeq I$ as opposed to $X \succeq J$. If you want to express a constraint $X \succeq \alpha J$ where J is a matrix of all ones, use the new `picos.J`.

Added

- Support for the OSQP solver.
- On-the-fly loading of `scipy.sparse` matrices. (See new note *NumPy and SciPy*.)
- Ability to negate or scale any expression and to sum any two expressions with the same or with a different type. This is established through a new `WeightedSum` fallback class. Convex or concave weighted sums can be used as an objective or in a constraint like any other expression.
- Properties `sp`, `np` and `np2d` to query the value of an expression as a SciPy or NumPy type. (See new class `Valuable` for all value query options.)
- Ability to use `numpy.array` directly on valued PICOS objects, returning a zero, one or two-dimensional array depending on the shape of the value.
- New method `require` and an equivalent overload for += to add constraints to a `Problem`.
- Cached functions `I`, `J`, and `0` that create, respectively, an identity matrix, a matrix of all ones, and a zero matrix.
- Cached properties `BiaffineExpression.rowsum` and `colsum` to complement the existing property `sum` and an argument `axis` to `picos.sum` for the same purpose.
- Option to give a name to `problems` via the first initialization argument or the `name` property.
- Ability to perform some algebraic operations on `objectives`.
- Support for solving nonconvex continuous quadratic programs (QPs) with CPLEX and Gurobi. Gurobi further allows convex quadratic constraints to be present.
- Ability to `reshape` affine expressions in C-order, like NumPy.
- Ability to pass constant values to `picos.sum`, `min` and `max`.

- Global option `settings.RETURN_SOLUTION` that controls whether `solve` returns a `Solution`.
- Methods `Samples.shuffled` and `kfold`.
- Support for MOSEK remote optimization with the *mosek_server* option.
- Option *cplex_vmconfig* to load a virtual machine configuration file with CPLEX.
- Function `picos.patch_scipy_array_priority` to work around SciPy#4819.

Changed

- The performance of solving semidefinite programs with trivial linear matrix inequalities of the form `X >> 0` using MOSEK (Optimizer) has been improved dramatically. Depending on your problem, you might experience this speedup when using the *dualize* option.
- `Problem.minimize` and `Problem.maximize` are now properties that you can assign a minimization or maximization objective to, respectively.
- All expression types as well as the classes `Problem` and `Objective` now share the same interface to query their (objective) value. In particular, the new `np` property can be used on all.
- Solving with `duals=True` will now raise an exception when duals were returned by the solver but not all could be converted. Use the default of `duals=None` to accept also incomplete duals.
- The new argument `name` is the only optional argument to `Problem` that may be passed as a positional argument; the arguments `copyOptions` and `useOptions` must now be passed as keyword arguments.

Fixed

- Running `setup.py` under Python 3.6 and earlier.
- Bad shebang lines; all are now properly reading `#!/usr/bin/env python3`.
- Incorrect duals returned by MOSEK (Fusion).
- An assertion failure when multiplying some quadratic expressions with a negative scalar.
- A false expression being created when multiplying a `DetRootN` with a negative scalar.
- An exception when multiplying a scalar power with a constant.
- A modify-during-iteration issue that could result in a suboptimal solver being chosen.
- Building piecewise affine functions from a mix of certain and random expressions.
- A failure when computing the convex hull of a `ScenarioPerturbationSet` with few points.
- Detection of string groups where the variable part is at the start or end of the strings.
- CVXOPT reacting inconsistently to some infeasible problems.
- A potential variable clash when reformulating a `NuclearNormConstraint`.
- Grammatical issues when printing variable groups of a problem.

Removed

- The deprecated functions *Problem.minimize* and *Problem.maximize*. See **Changed** for the new meaning of these names.
- The deprecated arguments `it` and `indices` to *picos.sum*.

7.4 2.2 - 2021-02-09

The Python 3 update.

Important

- PICOS now requires Python 3.4 or later; Python 2 support was dropped.

Added

- A synopsis to the *NoStrategyFound* exception, explaining why strategy search failed.

Fixed

- Optimizing matrix (p, q)-norms when columns of the matrix are constant.
- Refining norms over a sparse constant term to a constant affine expression.
- Gurobi printing empty lines to console when dual retrieval fails.

Changed

- A bunch of Python 2 compatibility code was finally removed.
- Exception readability has been improved using Python 3's `raise from` syntax where applicable.
- The `__version_info__` field now contains integers instead of strings.
- *QuadraticExpression.scalar_factors* is now `None` instead of an empty tuple when no decomposition into scalar factors is known.

Deprecated

- *QuadraticExpression.quadratic_forms*, as write access would leave the expression in an inconsistent state. (At your own risk, use the equivalent `_sparse_quads` instead.)

7.5 2.1 - 2020-12-29

The robust optimization update.

Important

- The sign of dual values for affine equality constraints has been fixed by inversion.

Added

- Support for a selection of robust optimization (RO) and distributionally robust stochastic programming (DRO) models through a new `picos.uncertain` namespace. You may now solve
  - scenario-robust conic programs via `ScenarioPerturbationSet`,
  - conically robust linear programs and robust conic quadratic programs under ellipsoidal uncertainty via `ConicPerturbationSet` and `UnitBallPerturbationSet`, and
  - least squares and piecewise linear stochastic programs where the data generating distribution is defined ambiguously through a Wasserstein ball or through bounds on its first two moments via `WassersteinAmbiguitySet` and `MomentAmbiguitySet`, respectively.
- New function `picos.block` to create block matrices efficiently.
- New convenience class `picos.Samples` for data-driven applications.
- New set class `picos.Ellipsoid` (has overlap with but a different scope than `picos.Ball`).
- Support for `matrix reshuffling` (aka *matrix realignment*) used in quantum information theory.
- Ability to define cones of fixed dimensionality and `product cones` thereof.
- Ability to query the `solver-reported objective value` (useful with RO and DRO objectives).
- Methods `Problem.conic_form` and `reformulated` for internal use and educational purposes.
- New module `picos.settings` defining global options that can be set through environment variables prefixed with `PICOS_`. Among other things, you can now blacklist all proprietary solvers for an application by passing `PICOS_NONFREE_SOLVERS=False` to the Python interpreter.
- A new base class `BiaffineExpression` for all (uncertain) affine expression types. This gives developers extending PICOS a framework to support models with parameterized data.
- Support for `factoring out` variables and parameters from (bi)affine vector expression.
- Support for `replacing` variables and parameters with affine expressions of same shape to perform a change of variables in a mathematical sense.
- Support for SCIP Optimization Suite 7.
- CVXOPT-specific solution search options *cvxopt_kktsolver* and *cvxopt_kktreg*.

Fixed

- Quadratic expressions created from a squared norm failing to decompose due to a numerically singular quadratic form.
- Solution objects unintendedly sharing memory.
- Solution search options that take a dictionary as their argument.
- Solution search with *assume_conic* set to `False`.
- The `EpigraphReformulation` falsely claiming that it can reformulate any nonconvex objective.
- A division by zero that could occur when computing the solution search overhead.
- An exception with functions that look for short string descriptions, in particular with `picos.sum`.

Changed

- The functions *picos.max* and *picos.min* can now be used to express the maximum over a list of convex and the minimum over a list of concave expressions, respectively.
- Squared norms are now implemented as a subclass of quadratic expressions (*SquaredNorm*), skipping an unnecessary decomposition on constraint creation.
- Commutation matrices used internally for various algebraic tasks are now retrieved from a centralized cached function, improving performance.
- The string description of *Problem* instances is not enclosed by dashed lines any more.

7.6 2.0 - 2020-03-03

The backend update.

Important

This is a major release featuring vast backend rewrites as well as interface changes. Programs written for older versions of PICOS are expected to raise deprecation warnings but should otherwise work as before. The following lists notable exceptions:

- The solution returned by *solve* is now an instance of the new *Solution* class instead of a dictionary.
- If solution search fails to find an optimal primal solution, PICOS will now raise a *SolutionFailure* by default. Old behavior of not raising an exception is achieved by setting `primals=None` (see *primals* and *duals* options).
- The definition of the $L_{p,q}$-norm has changed: It no longer refers to the p-norm of the q-norms of the matrix rows but to the q-norm of the p-norms of the matrix columns. This matches the definition you would find on Wikipedia and should reduce confusion for new users. See *Norm*.
- The signs in the Lagrange dual problem of a conic problem are now more consistent for all cones, see *Dual Values*. In particular the signs of dual values for (rotated) second order conic constraints have changed and the problem obtained by *Problem.dual* (new for *as_dual*) has a different (but equivalent) form.

Added

- A modular problem reformulation framework. Before selecting a solver, PICOS now builds a map of problem types that your problem can be reformulated to and makes a choice based on the expected complexity of the reposed problem.
- An object oriented interface to solution search options. See *Options*.
- Support for arbitrary objective functions via an epigraph reformulation.
- Support for MOSEK 9.
- Support for ECOS 2.0.7.
- Support for multiple subsystems with *partial_trace*.
- Quick-solve functions *picos.minimize* and *picos.maximize*.
- Lower and upper diagonal matrix variable types.
- *SecondOrderCone* and *RotatedSecondOrderCone* sets to explicitly create the associated constraints. *(You now need to use these if you want to obtain a conic instead of a quadratic dual.)*
- Possibility to use *picos.sum* to sum over the elements of a single multidimensional expression.
- Possibility to create a *Ball* or *Simplex* with a non-constant radius.

- Many new properties (postfix operations) to work with affine expressions; for instance `A.vec` is a faster and cached way to express the vectorization `A[:]`.
- Options *assume_conic* and *verify_prediction*.
- An option for every solver to manipulate the chances of it being selected (e.g. *penalty_cvxopt*).
- Ability to run doctests via `test.py`.

Fixed

The following are issues that were fixed in an effort of their own. If a bug is not listed here, it might still be fixed as a side effect of some of the large scale code rewrites that this release ships.

- Upgrading the PyPI package via pip.
- A regression that rendered the Kronecker product unusable.
- Noisy exception handling in a sparse matrix helper function.
- Shape detection for matrices given by string.
- The *hotstart* option when solving with CPLEX.
- Low precision QCP duals from Gurobi.

Changed

- All algebraic expression code has been rewritten and organized in a new `expressions` package. In particular, real and complex expressions are distinguished more clearly.
- All algebraic expressions are now immutable.
- The result of any unary operation on algebraic expressions (e.g. negation, transposition) is cached (only computed once per expression).
- Slicing of affine expressions is more powerful, see *Matrix Slicing*.
- Loading of constant numeric data has been unified, see `load_data`.
- Variables are now created independently of problems by instanciating one of the new `variable types`. *(`Problem.add_variable` is deprecated.)*
- Constraints are added to problems as they are; any transformation is done transparently during solution search.
- In particular, $x^2 \leq yz$ is now initially a (nonconvex) quadratic constraint and transformation to a conic constraint is controlled by the new *assume_conic* option.
- Expressions constrained to be positive semidefinite are now required to be symmetric/hermitian by their own definition. *(Use `SymmetricVariable` or `HermitianVariable` whenever applicable!)*
- Options passed to `solve` are only used for that particular search.
- The default value for the *verbosity* option (formerly `verbose`) is now 0.
- Available solvers are only imported when they are actually being used, which speeds up import of PICOS on platforms with many solvers installed.
- The code obeys PEP 8 and PEP 257 more strongly. Exceptions: D105, D203, D213, D401, E122, E128, E221, E271, E272, E501, E702, E741.
- Production testing code was moved out of the `picos` package.

Removed

- The NoAppropriateSolverError exception that was previously raised by *solve*. This is replaced by the new *SolutionFailure* exception with error code 1.
- Some public functions in the *tools* module that were originally meant for internal use.

Deprecated

This section lists deprecated modules, functions and options with their respective replacement or deprecation reason on the right hand side. Deprecated entities produce a warning and will be removed in a future release.

- The *tools* module as a whole. It previously contained both algebraic functions for the user as well as functions meant for internal use. The former group of functions can now be imported directly from the *picos* namespace (though some are also individually deprecated). The other functions were either relocated (but can still be imported from *tools* while it lasts) or removed.
- In the *Problem* class:
  - *add_variable*, *remove_variable*, *set_var_value* → variables are instanciated directly and added to problems automatically
  - *minimize* → *picos.minimize*
  - *maximize* → *picos.maximize*
  - *set_option* → assign to attributes or items of *Problem.options*
  - *update_options* → *options.update*
  - *set_all_options_to_default* → *options.reset*
  - *obj_value* → *value*
  - *is_continuous* → *continuous*
  - *is_pure_integer* → *pure_integer*
  - *verbosity* → *options.verbosity*
  - *as_dual* → *dual*
  - *countVar*, *countCons*, *numberOfVars*, *numberLSEConstraints*, *numberSDPConstraints*, *numberQuadConstraints*, *numberConeConstraints* → were meant for internal use
  - arguments it, indices and key to *add_list_of_constraints* → are ignored
- All expression types:
  - constraint creation via < → <=
  - constraint creation via > → >=
  - *is_valued* → *valued*
  - *set_value* → assign to *value*
- Affine expressions:
  - *fromScalar* → *from_constant* or *picos.Constant*
  - *fromMatrix* → *from_constant* or *picos.Constant*
  - *hadamard* → ^
  - *isconstant* → *constant*
  - *same_as* → *equals*
  - *transpose* → *T*
  - *Tx* → *partial_transpose*

  - *conjugate* → *conj*
  - *Htranspose* → *H*
  - *copy* → expressions are immutable
  - *soft_copy* → expressions are immutable
- Algebraic functions and shorthands in the `picos` namespace:
  - *tracepow* → *PowerTrace*
  - *new_param* → *Constant*
  - *flow_Constraint* → *FlowConstraint*
  - *diag_vect* → *maindiag*
  - *simplex* → *Simplex*
  - *truncated_simplex* → *Simplex*
  - arguments `it` and `indices` to *sum* → are ignored
- Solution search options:
  - `allow_license_warnings` → *license_warnings*
  - `verbose` → *verbosity* (takes an integer)
  - `noprimals` → *primals* (the meaning is inverted)
  - `noduals` → *duals* (the meaning is inverted)
  - `tol` → `*_fsb_tol` and `*_ipm_opt_tol`
  - `gaplim` → *rel_bnb_opt_tol*
  - `maxit` → *max_iterations*
  - `nbsol` → *max_fsb_nodes*
  - `pool_relgap` → *pool_rel_gap*
  - `pool_absgap` → *pool_abs_gap*
  - `lboundlimit` → *cplex_lwr_bnd_limit*
  - `uboundlimit` → *cplex_upr_bnd_limit*
  - `boundMonitor` → *cplex_bnd_monitor*
  - `solve_via_dual` → *dualize* (may not be `None` any more)

7.7 1.2.0 - 2019-01-11

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.

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.

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.

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.

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.

7.8 1.1.3 - 2018-10-05

Added

- Support for the solvers GLPK and SCIP.
- PICOS packages on Anaconda Cloud.
- PICOS packages in the Arch Linux User Repository.

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.

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.

7.9 1.1.2 - 2016-07-04

Added

- Ability to dynamically add and remove constraints.
- Option `pass_simple_cons_as_bound`, see below.

Changed

- Improved efficiency when processing large expressions.
- Improved support for the SDPA solver.
- `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.

Fixed

- Minor bugs with complex expressions.

7.10 1.1.1 - 2015-08-29

Added

- Support for the SDPA solver.
- Partial trace of an affine expression, see `partial_trace`.

Changed

- Improved PEP 8 compliance.

Fixed

- Compatibility with Python 3.

7.11 1.1.0 - 2015-04-15

Added

- Compatibility with Python 3.

Changed

- The main repository has moved to GitHub.

7.12 1.0.2 - 2015-01-30

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.
- Support for antisymmetric matrix variables.
- 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*.

Changed

- Improved efficiency of the sparse SDPA file format writer.
- Improved efficiency of the complex to real transformation.

Fixed

- Scalar product of hermitian matrices.
- Conjugate of a complex expression.

7.13 1.0.1 - 2014-08-27

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 argument to *tracepow*, to represent constraints of the form $\operatorname{trace}(MX^p) \geq t$.

Changed

- Significantly improved slicing performance for affine expressions.
- Improved performance when loading data.
- Improved performance when retrieving primal solution from CPLEX.
- The documentation received an overhaul.

7.14 1.0.0 - 2013-07-19

Added

- Ability to express rational powers of affine expressions with the `**` operator, traces of matrix powers with *tracepow*, (generalized) p-norms with *norm* and n-th roots of a determinant with *detrootn*.
- Ability to specify variable bounds directly rather than by adding constraints, see *add_variable*.
- Problem dualization.
- 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.

Changed

- If the chosen solver supports this, updated problems will be partially re-solved instead of solved from scratch.

Removed

- Option `onlyChangeObjective`.

7.15 0.1.3 - 2013-04-17

Added

- A *`geomean`* function to construct geometric mean inequalities that will be cast as rotated second order cone 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.

Changed

- The solver search time is returned in the dictionary returned by *`solve`*.

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`*.

7.16 0.1.2 - 2013-01-10

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`*.

7.17 0.1.1 - 2012-12-08

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.

Fixed

- Minor bugs with quadratic expressions.

7.18 0.1.0 - 2012-06-22

Added

- Initial release of PICOS.

CHAPTER

EIGHT

CONTRIBUTION GUIDE

8.1 Filing a bug report or feature request

Via GitLab

If you have a GitLab account, just head to PICOS' official issue tracker.

Via mail

If you don't have a GitLab account you can still create an issue by writing to incoming+picos-api/picos@incoming.gitlab.com. Unlike issues created directly on GitLab, issues created by mail are *not* publicly visible.

8.2 Submitting a code change

The canonical 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.

8.3 Code style

Set your linter to enforce **PEP 8** and **PEP 257** except for the following codes:

```
D105,D203,D213,D401,E122,E128,E221,E271,E272,E501,E702,E741
```

Our line width limit is `80` characters.

8.4 Release procedure

Version scheme

When installed from the git repository or from a source distribution (sdist), PICOS versions have the format `MAJOR.MINOR.PATCH`, where `PATCH` is the commit distance to the last minor release. When installed from a source tarball that carries no metadata from either git or setuptools, the version format is just `MAJOR.MINOR` as the commit distance defining the `PATCH` bit is not known. Note that the `PATCH` bit is not strictly incremental as not every commit to the `master` branch is released individually.

Bumping the version

To bump the major or minor version, run `version.py -b MAJOR.MINOR`. This commits that base version to the `picos/.version` file and creates an annotated tag (`vMAJOR.MINOR`) for that commit. The release of version `MAJOR.MINOR.0` is then published by pushing the tagged commit to the top of `origin/master`, which triggers a GitLab CI/CD pipeline for that commit. By the same logic, the `PATCH` bit is bumped and the resulting version is published automatically whenever a number of commits is pushed to the `master` branch between two minor versions.

Note that source distributions are aware of the `PATCH` bit as setuptools writes it to the `picos/.version` file in the source tree.

Justification

The result of this unorthodox release procedure is that bugfix releases can be made quickly simply by pushing a commit to `master`. On the other hand, changes that should go into the next minor or major release must remain on topic branches and pushed onto `master` together with the commit from `version.py -b`.

8.5 Implementing a test case

Production test sets are implemented in the files in the `tests` folder that start with `ptest_`. 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.

8.6 Implementing a solver

If you want to implement support for a new solver, all you have to do is update `solvers/__init__.py` where applicable, and add a file named `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 of 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`.

BIBLIOGRAPHY

[svec] Dattorro, J. (2018). Isomorphism of symmetric matrix subspace. In *Convex Optimization & Euclidean Distance Geometry (2nd ed.)* (pp. 47f.). California, Meboo Publishing USA. Retrieved from https://meboo.convexoptimization.com/Meboo.html.

PYTHON MODULE INDEX

INDEX