MOSEK Fusion API for Python

Release 8.0.0.74

MOSEK ApS

2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

The **MOSEK** Optimization Suite 8.0.0.74 is a powerfull software package capable of solving large-scale optimization problems of the following kind:

- linear,
- convex quadratic,
- conic quadratic (also known as second-order cone),
- semidefinite,
- and general convex.

Integer constrained variables are supported for all problem classes except for semidefinite and general convex problems. In order to obtain an overview of features in the **MOSEK** Optimization Suite consult the product introduction guide.

1.1 Why the Fusion API for Python?

Fusion is an object oriented API specifically designed to build conic optimization models in a simple and expressive manner, using mainstream programming languages.

With focus on usability and compactness, it helps the user focus on the modeling instead of coding.

The most widespread class of optimization problems is *linear optimization problems*, where all relations are linear. The tremendous success of both applications and theory of linear optimization can be ascribed to the following factors:

- The required data are simple, i.e. just matrices and vectors.
- Convexity is guaranteed since the problem is convex by construction.
- Linear functions are trivially differentiable.
- There exist very efficient algorithms and software for solving linear problems.
- Duality properties for linear optimization are nice and simple.

Even if the linear optimization model is only an approximation to the true problem at hand, the many advantages of linear optimization may outweight the disadvantages. In some cases, however, the problem formulation is inherently nonlinear and a linear approximation is either intractable or inadequate. *Conic* *optimization* has proved to be a very expressive and powerful way to introduce nonlinearities, while preserving all the nice properties of linear optimization listed above.

The fundamental expression in linear optimization is a linear expression of the form

$$Ax - b \in \mathcal{K}$$

where $\mathcal{K} = \{y : y \geq 0\}$, i.e.,

$$\begin{aligned} Ax - b &= y, \\ y &\in \mathcal{K}. \end{aligned}$$

In conic optimization a wider class of convex sets $\mathcal{K}$ is allowed, for example in 3 dimensions $\mathcal{K}$ may correspond to an ice cream cone. The conic optimizer in **MOSEK** supports three structurally different types of cones $\mathcal{K}$, which allows a surprisingly large number of nonlinear relations to be modeled (as described in the **MOSEK** modeling cookbook), while preserving the nice algorithmic and theoretical properties of linear optimization.

A typical (low-level) solver API requires the problem to be serialized into a single matrix and a few vectors, and constructing (or modifying) such a problem often proves to be both a time-consuming and error-prone process. *Fusion*, on the other hand, introduces a higher level of abstraction, which allows the user to focus explicitly on modeling oriented aspects rather than reformulating a given model for a particular solver API. For example, in *Fusion* it is easy to add variables and constraints to an existing model.

Typically a conic optimization model in *Fusion* can be developed in a fraction of the time compared to using a low-level C API, but of course *Fusion* introduces a computational overhead compared to customized C code. In most cases, however, the overhead is small compared to the overall solution time, and we generally recommend that *Fusion* is used as a first step for building and verifying new models. Often, the final *Fusion* implementation will be directly suited for production code, and otherwise it readily provides a reference implementation for model verification.

1.2 License agreement

Before using the **MOSEK** software, please read the license agreement available in the distribution at `<MSKHOME>/mosek/8/mosek-eula.pdf` or on the **MOSEK** website https://mosek.com/sales/license-agreement.

MOSEK uses some third-party open-source libraries. Their license details follows.

zlib

MOSEK includes the *zlib* library obtained from the zlib website. The license agreement for *zlib* is shown in Listing 1.1.

Listing 1.1: *zlib* license.

```
zlib.h -- interface of the 'zlib' general purpose compression library
version 1.2.7, May 2nd, 2012

Copyright (C) 1995-2012 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

fplib

MOSEK includes the floating point formatting library developed by David M. Gay obtained from the netlib website. The license agreement for *fplib* is shown in Listing 1.2.

Listing 1.2: *fplib* license.

CHAPTER

TWO

INSTALLATION

In this section we discuss how to install and setup the **MOSEK** Fusion API for Python.

2.1 Compatibility

Fusion API requires Python 2.7, 3.5 or later and `numpy`.

2.2 Instructions

To install *Fusion* run the `setup.py` script located in <MSKHOME>/mosek/8/tools/platform/<PLATFORM>/python/2 or <MSKHOME>/mosek/8/tools/platform/<PLATFORM>/python/3 depending on the Python version you want to use. For instance, to install **MOSEK** for Python 3 on a user level, i.e., with no special system privilegdes needed, type

```
$ python3 <MSKHOME>/mosek/8/tools/platform/<PLATFORM>/python/3/setup.py install --user
```

where

- <PLATFORM> is either *linux64x86* or *osx64x86*.
- <MSKHOME> is the root of the **MOSEK** installation.

2.3 Running Examples

The examples directories contain scripts for building and running the various examples; these are called `build.bat` and `run.bat` on Windows, and `build.sh` and `run.sh` on Linux and Mac OS X. Running all examples and making sure that all complete with no error is a recommeneded preliminary step to check that **MOSEK** *Fusion* is properly set up.

Note: A valid license must be available and set up.

CHAPTER

THREE

GUIDELINES

3.1 Known Limitations

The main limitation in the use of the **MOSEK** Fusion API for Python 8.0.0.74 are reported in this section.

3.1.1 Modeling Limitations

To design an API that looks almost the same across several programming languaes, some limitations are needed:

Fusion imposes some limitations on certain aspects of a model:

- Constraints and variables belong to a single model, and cannot as such be used (e.g. stacked) with objects from other models.
- Constraint and variable domains are immutable.

3.1.2 Memory Limitations

There are some hard limits on shapes and sizes in *Fusion*:

- The maximum number of variable elements used in a model can be no larger than $2^{31} - 1$.
- The maximum size of a dimension is $2^{31} - 1$.
- For efficiency reasons the total size of an item (the product of the dimensions) is limited to $2^{63} - 1$.

Fetching a solution from a shaped variable produces a flat array of values. This means that all values, even the ones that are not used in the problem, are returned, and that the variable elements are linearly indexed. In this case, it is better to create a slice variable holding the relevant elements and fetch the solution for this; fetching the full solution may cause an exception due to memory exhaustion or platform-dependant constraints on array sizes.

Users might experience memory leaks using *Fusion*:

- memory usage not decreasing after the solver terminates.
- memory usage increasing during the resolution of a sequence of problems.

In both cases the reason might be due to **MOSEK** not releasing all the memory allocated by the `Model` class. Each `Model` object links to an external resource (the **MOSEK** task). Due to the way the garbage collector works it is not guaranteed when the `Model` object can be reclaimed, and in some cases it cannot be automatically reclaimed at all. This means that substantial amounts of memory may be leaked.

For this reason it is very important to always make sure that the `Model` object is properly disposed of when it is not used anymore by calling `Model.__del__()`.

Despite this operation might seems redundant in some situations, i.e. when a single problem of moderate size must be solved, it is a good practice to always take care of the memory management. This simple steps will result in a more robust code, and less probability of future memory issues. The *Model* supports the *Context Manager* protocol, which means that by using the construction

```
with MyModel() as M:
    pass;
```

ensures that `__del__()` is called when the with-scope ends, even if an exception was raised. If this cannot be used, e.g. if the *Model* object is returned by a factory function, the method must be explicitly called. Furthermore, if the *Model* class is inherited and any work is done in the constructor, it is necessary to ensure that if an exception is thrown during construction, the `Model.__del__()` is called. To ensure this, use a construction along the lines

```
import mosek
from mosek.fusion import *

class MyModel(Model):
    def __init__(self):
        Model.__init__(self)
        finished = False
        try:
            finished = True
        finally:
            if not finished:
                self.__del__()

with MyModel() as M:
    pass;
```

Note that this construction avoids having to catch and rethrow an exception in the constructor, thus changing the exception stack trace.

3.2 Deployment

When redistributing a Python application using the **MOSEK** Fusion API for Python 8.0.0.74, the following libraries must be included:

64-bit Linux	64-bit Windows	32-bit Windows	64-bit Mac OS
`libmosek64.so.8.0`	`mosek64_8_0.dll`	`mosek32_8_0.dll`	`libmosek64.dylib.8.0`
`libiomp5.lib`	`libiomp5md.dll`	`libomp5md.dll`	`libiomp5.dylib`
`libcilkrts.so.5`	`cilkrts20.dll`	`cilkrts20.dll`	`libcilkrts.5.dylib`
`libmosekxx8_0.so`	`mosekxx8_0.dll`	`mosekxx8_0.dll`	`libmosekxx8_0.dylib`

Furthermore, one (or both) of the directories

- `python/2/mosek` for Python 2.x applications, and
- `python/3/mosek` for Python 3.x applications.

must be included.

By default the **MOSEK** Python API will look for the binary libraries in the **MOSEK** module directory, i.e. the directory containing `__init__.py`. Alternative, if the binary libraries reside in another directory, the application can pre-load the `mosekxx` library from another located before `mosek` is imported, e.g. like this

```
import ctypes ; ctypes.CDLL('my/path/to/mosekxx.dll')
```

3.3 The license system

MOSEK is a commercial product that **always** needs a valid license to work. A license is typically provided as a license file that allows the user to access the subset of the **MOSEK** Optimization Suite functionalities it is entitled for, and for the right amount of time. **MOSEK** uses a third party license manager to implement license checking.

By default a license token remains checked out for the duration of the **MOSEK** session, i.e.

1. a license token is checked out when the method *Model.solve* is called the first time and
2. it is returned when the *Model* class instance is destroyed.

To change the license systems behavior to returning the license token after each call to **MOSEK** set the parameter *cacheLicense* to `off`.

Additionally license checkout and checkin can be controlled manually accessing the unerlying **MOSEK** task and environment. Please see Section *8.4*.

3.3.1 Waiting for a free license

By default an error will be returned if no license token is available. By setting the parameter *licenseWait* **MOSEK** can be instructed to wait until a license token is available.

See section *8.1*.

3.3.2 Manually stopping the license system

CHAPTER

FOUR

BASIC TUTORIALS

In this section a number of examples is provided to demonstrate the functionality required for solving linear, conic, semidefinite and quadratic problems as well as mixed integer problems.

- *Linear optimization tutorial* : It shows how to input a linear program. It will show how
  - define variables and their bounds,
  - define constraints and their bounds,
  - define a linear objective function,
  - input a linear program but rows or by column.
  - retrieve the solution.
- *Conic quadratic optimization tutorial* : The basic steps needed to formulate a conic quadratic program are introduced:
  - define quadratic cones,
  - assign the relevant variables to their cones.
- *Semidefinite optimization tutorial* : How to input semidefintite optimization problems is the topic of this tutorial, and in particular how to
  - input semidefinite matrices and in sparse format,
  - add semidefinite matrix variable and
  - formulate linear constraints and objective function based on matrix variables.
- *Mixed-Integer optimization tutorial* : This tutorial shows how integrality conditions can be specified.

4.1 Linear Optimization

The simplest optimization problem is a purely linear problem. A *linear optimization problem* is a problem of the following form:

Minimize or maximize the objective function

$$\sum_{j=0}^{n-1} c_j x_j + c^f$$

subject to the linear constraints

$$l_k^c \leq \sum_{j=0}^{n-1} a_{kj} x_j \leq u_k^c, \quad k = 0, \ldots, m-1,$$

and the bounds

$$l_j^x \leq x_j \leq u_j^x, \quad j = 0, \ldots, n-1,$$

where we have used the problem elements:

- m and n which are the number of constraints and variables respectively,
- x which is the variable vector of length n,
- c which is a coefficient vector of size n

$$c = \begin{bmatrix} c_0 \\ \vdots \\ c_{n-1} \end{bmatrix},$$

- c^f which is a constant,
- A which is a $m \times n$ matrix of coefficients is given by

$$A = \begin{bmatrix} a_{0,0} & \cdots & a_{0,(n-1)} \\ \vdots & \cdots & \vdots \\ a_{(m-1),0} & \cdots & a_{(m-1),(n-1)} \end{bmatrix},$$

- l^c and u^c which specify the lower and upper bounds on constraints respectively, and
- l^x and u^x which specifies the lower and upper bounds on variables respectively.

Note: Please note the unconventional notation using 0 as the first index rather than 1. Hence, x_0 is the first element in variable vector x.

4.1.1 Example LO1

The following is an example of a linear optimization problem:

$$\begin{array}{lccccccccc} \text{maximize} & 3x_0 & + & 1x_1 & + & 5x_2 & + & 1x_3 & & \\ \text{subject to} & 3x_0 & + & 1x_1 & + & 2x_2 & & & = & 30, \\ & 2x_0 & + & 1x_1 & + & 3x_2 & + & 1x_3 & \geq & 15, \\ & & & 2x_1 & & & + & 3x_3 & \leq & 25, \end{array} \tag{4.1}$$

having the bounds

$$\begin{array}{ccccc} 0 & \leq & x_0 & \leq & \infty, \\ 0 & \leq & x_1 & \leq & 10, \\ 0 & \leq & x_2 & \leq & \infty, \\ 0 & \leq & x_3 & \leq & \infty. \end{array}$$

We start our implementation in *Fusion* importing the relevant modules, i.e.

```
from  mosek.fusion import *
```

Next we declare an optimization model creating an instance of the *Model* class:

```
with Model("lo1") as M:
```

From now on most of our steps will involve `M`. The variables in problem (4.1) can be declared specifying:

- an (optional) name,
- their dimension,
- the bounds.

```
# Create variable 'x' of length 3
x = M.variable("x", 4, Domain.greaterThan(0.0))
```

It is important to notice that the bound will be applied element-wise.

To define the constraints, we assume the coeffient matrix to be given as an array of rows A, each one being a dense array as well.

```
# Create three constraints
M.constraint("c1", Expr.dot(A[0], x), Domain.equalsTo(30.0))
M.constraint("c2", Expr.dot(A[1], x), Domain.greaterThan(15.0))
M.constraint("c3", Expr.dot(A[2], x), Domain.lessThan(25.0))
```

We end the definition of our optimization model setting the objective function: the coefficient are assumed to be given in a single one dimensional array c.

```
# Set the objective function to (c^t * x)
M.objective("obj", ObjectiveSense.Maximize, Expr.dot(c, x))
```

Finally, we only need to call the `Model.solve` method:

```
M.solve()
```

The values attained by each variable can be obtained using the `Variable.level` method.

The complete code follows in Listing 4.1.

Listing 4.1: *Fusion* implementation of model (4.1).

```
import sys

import mosek
import mosek.fusion
from  mosek.fusion import *

def main(args):
    A = [ [ 3.0, 2.0, 0.0, 1.0 ],
          [ 2.0, 3.0, 1.0, 1.0 ],
          [ 0.0, 0.0, 3.0, 2.0 ] ]
    c = [ 3.0, 5.0, 1.0, 1.0  ]

    # Create a model with the name 'lo1'
    with Model("lo1") as M:

      # Create variable 'x' of length 3
      x = M.variable("x", 4, Domain.greaterThan(0.0))

      # Create three constraints
      M.constraint("c1", Expr.dot(A[0], x), Domain.equalsTo(30.0))
      M.constraint("c2", Expr.dot(A[1], x), Domain.greaterThan(15.0))
      M.constraint("c3", Expr.dot(A[2], x), Domain.lessThan(25.0))

      # Set the objective function to (c^t * x)
      M.objective("obj", ObjectiveSense.Maximize, Expr.dot(c, x))

      # Solve the problem
      M.solve()


      # Get the solution values
      sol = x.level()
      print('\n'.join(["x[%d] = %f"%(i, sol[i]) for i in range(4)]) )
```

```
if __name__ == '__main__':
    main(sys.argv[1:])
```

4.2 Conic Quadratic Optimization

Conic optimization is a generalization of linear optimization, allowing constraints of the type

$$x^t \in \mathcal{K}_t,$$

where x^t is a subset of the problem variables and $\mathcal{K}_t$ is a convex cone. Actually, since the set $\mathbb{R}^n$ of real numbers is also a convex cone, all variables can in fact be partitioned into subsets belonging to separate convex cones, simply stated $x \in \mathcal{K}$.

MOSEK can solve conic quadratic optimization problems of the form

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & x \in \mathcal{K}, & & \end{array}$$

where the domain restriction, $x \in \mathcal{K}$, implies that all variables are partitioned into convex cones

$$x = (x^0, x^1, \ldots, x^{p-1}), \quad \text{with } x^t \in \mathcal{K}_t \subseteq \mathbb{R}^{n_t}.$$

For convenience, the user only specify subsets of variables x^t belonging to cones $\mathcal{K}_t$ different from the set $\mathbb{R}^{n_t}$ of real numbers. These cones can be a:

- Quadratic cone:

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n : x_0 \geq \sqrt{\sum_{j=1}^{n-1} x_j^2} \right\}.$$

- Rotated quadratic cone:

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n : 2x_0 x_1 \geq \sum_{j=2}^{n-1} x_j^2, \quad x_0 \geq 0, \quad x_1 \geq 0 \right\}.$$

From these definition it follows that

$$(x_4, x_0, x_2) \in \mathcal{Q}^3,$$

is equivalent to

$$x_4 \geq \sqrt{x_0^2 + x_2^2}.$$

Furthermore, each variable may belong to one cone at most. The constraint $x_i - x_j = 0$ would however allow x_i and x_j to belong to different cones with same effect.

4.2.1 Example CQO1

We want to solve the following Conic Optimization Problem problem:

$$\begin{array}{ll} \min & y_1 + y_2 + y_3 \\ s.t. & \\ & x_1 + x_2 + 2.0 x_3 = 1.0 \\ & x_1, x_2, x_3 \geq 0.0 \\ & (y_1, x_1, x_2) \in \mathcal{Q}^3, \\ & (y_2, y_3, x_3) \in \mathcal{Q}_r^3 \end{array} \tag{4.2}$$

is an example of a conic quadratic optimization problem. The problem involves some linear constraints, a quadratic cone and a rotated quadratic cone.

We start creating the optimization model:

```
with Model('cqo1') as M:
```

We than define variables x and y, the former non negative, the latter free. Two logical variables *z1* and *z2* are introduced as they will be used to define the second order cones.

```
x = M.variable('x', 3, Domain.greaterThan(0.0))
y = M.variable('y', 3, Domain.unbounded())

# Create the aliases
#      z1 = [ y[0],x[0],x[1] ]
#  and z2 = [ y[1],y[2],x[2] ]
z1 = Var.vstack(y.index(0),  x.slice(0,2))
z2 = Var.vstack(y.slice(1,3),x.index(2))
```

It is important to note that *z1* and *z2* are just logical variables, i.e. just map onto x,y. They are introduced for convenience sake.

The linear constraint are defined simply multiplying an array of coefficients with x:

```
# Create the constraint
#      x[0] + x[1] + 2.0 x[2] = 1.0
M.constraint("lc", Expr.dot([1.0, 1.0, 2.0], x), Domain.equalsTo(1.0))
```

The conic constraints are defined using the logical views *z1* and *z2*:

```
# Create the constraints
#      z1 belongs to C_3
#      z2 belongs to K_3
# where C_3 and K_3 are respectively the quadratic and
# rotated quadratic cone of size 3, i.e.
#                 z1[0] > sqrt(z1[1]^2 + z1[2]^2)
#  and  2.0 z2[0] z2[1] > z2[2]^2
qc1 = M.constraint("qc1", z1, Domain.inQCone())
qc2 = M.constraint("qc2", z2, Domain.inRotatedQCone())
```

Note that this is not the only way to define that conic constraints. But it is in this case probably the cleanest and faster.

We only need our objective function:

```
# Set the objective function to (y[0] + y[1] + y[2])
M.objective("obj", ObjectiveSense.Minimize, Expr.sum(y))
```

Just call the *Model.solve* method to run the solver:

```
M.solve()
```

The solution can be retrieve using *Variable.level*, while the dual multipliers of the constraints are available via the *Variable.dual* method. For the linear part

```
# Get the linearsolution values
solx = x.level()
soly = y.level()
```

while the conic quadratic part can be retrieve easily as well.

```
# Get conic solution of qc1
qc1lvl = qc1.level()
qc1sn  = qc1.dual()
```

The complete code follows in Listing 4.2.

Listing 4.2: *Fusion* implementation of model (4.2).

```
import mosek
import mosek.fusion
from   mosek.fusion import *

with Model('cqo1') as M:

  x = M.variable('x', 3, Domain.greaterThan(0.0))
  y = M.variable('y', 3, Domain.unbounded())

  # Create the aliases
  #      z1 = [ y[0],x[0],x[1] ]
  #  and z2 = [ y[1],y[2],x[2] ]
  z1 = Var.vstack(y.index(0),  x.slice(0,2))
  z2 = Var.vstack(y.slice(1,3),x.index(2))

  # Create the constraint
  #      x[0] + x[1] + 2.0 x[2] = 1.0
  M.constraint("lc", Expr.dot([1.0, 1.0, 2.0], x), Domain.equalsTo(1.0))

  # Create the constraints
  #      z1 belongs to C_3
  #      z2 belongs to K_3
  # where C_3 and K_3 are respectively the quadratic and
  # rotated quadratic cone of size 3, i.e.
  #                 z1[0] > sqrt(z1[1]^2 + z1[2]^2)
  #  and  2.0 z2[0] z2[1] > z2[2]^2
  qc1 = M.constraint("qc1", z1, Domain.inQCone())
  qc2 = M.constraint("qc2", z2, Domain.inRotatedQCone())

  # Set the objective function to (y[0] + y[1] + y[2])
  M.objective("obj", ObjectiveSense.Minimize, Expr.sum(y))

  # Solve the problem
  M.solve()

  # Get the linearsolution values
  solx = x.level()
  soly = y.level()
  print ('x1,x2,x3 = %s' % str(solx))
  print ('y1,y2,y3 = %s' % str(soly))

  # Get conic solution of qc1
  qc1lvl = qc1.level()
  qc1sn  = qc1.dual()
  print ('qc1 levels                = %s' % str(qc1lvl))
  print ('qc1 dual conic var levels = %s' % str(qc1sn))
```

4.3 Semidefinite Optimization

Semidefinite optimization is a generalization of conic quadratic optimization, allowing the use of matrix variables belonging to the convex cone of positive semidefinite matrices

$$\mathcal{S}_+^r = \left\{ X \in \mathcal{S}^r : z^T X z \geq 0, \quad \forall z \in \mathbb{R}^r \right\},$$

where $\mathcal{S}^r$ is the set of $r \times r$ real-valued symmetric matrices.

MOSEK can solve semidefinite optimization problems of the form

$$\begin{array}{lccccl} \text{minimize} & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle + c^f & & & \\ \text{subject to} & l_i^c \le & \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \le & u_i^c, & i = 0, \dots, m-1, \\ & l_j^x \le & x_j & \le & u_j^x, & j = 0, \dots, n-1, \\ & & x \in \mathcal{K}, \overline{X}_j \in \mathcal{S}_+^{r_j}, & & & j = 0, \dots, p-1 \end{array}$$

where the problem has p symmetric positive semidefinite variables $\overline{X}_j \in \mathcal{S}_+^{r_j}$ of dimension r_j with symmetric coefficient matrices $\overline{C}_j \in \mathcal{S}^{r_j}$ and $\overline{A}_{i,j} \in \mathcal{S}^{r_j}$. We use standard notation for the matrix inner product, i.e., for $A, B \in \mathbb{R}^{m \times n}$ we have

$$\langle A, B \rangle := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij} B_{ij}.$$

4.3.1 Example SDO1

The problem

$$\begin{array}{llcl} \text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \overline{X} \right\rangle + x_0 & & \\ \text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_0 & = & 1, \\ & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_1 + x_2 & = & 1/2, \\ & x_0 \ge \sqrt{{x_1}^2 + {x_2}^2}, & & \overline{X} \succeq 0, \end{array} \tag{4.3}$$

is a mixed semidefinite and conic quadratic programming problem with a 3-dimensional semidefinite variable

$$\overline{X} = \begin{bmatrix} \overline{X}_{00} & \overline{X}_{10} & \overline{X}_{20} \\ \overline{X}_{10} & \overline{X}_{11} & \overline{X}_{21} \\ \overline{X}_{20} & \overline{X}_{21} & \overline{X}_{22} \end{bmatrix} \in \mathcal{S}_+^3,$$

and a conic quadratic variable $(x_0, x_1, x_2) \in \mathcal{Q}^3$. The objective is to minimize

$$2(\overline{X}_{00} + \overline{X}_{10} + \overline{X}_{11} + \overline{X}_{21} + \overline{X}_{22}) + x_0,$$

subject to the two linear constraints

$$\overline{X}_{00} + \overline{X}_{11} + \overline{X}_{22} + x_0 = 1,$$

and

$$\overline{X}_{00} + \overline{X}_{11} + \overline{X}_{22} + 2(\overline{X}_{10} + \overline{X}_{20} + \overline{X}_{21}) + x_1 + x_2 = 1/2.$$

$$\begin{array}{ll} \min & Tr\left(\overline{C} \cdot \overline{X}\right) + x_0 \\ s.t. & \\ & Tr(\overline{A}_0 \cdot \overline{X}) + x_0 = 1.0 \\ & Tr(\overline{A}_1 \cdot \overline{X}) + x_1 + x_2 = \frac{1}{2} \\ & (x_0, x_1 x_2) \in \mathcal{Q}^3 \\ & \overline{X} \in \mathcal{S}_+, \end{array} \tag{4.4}$$

where

$$\overline{C} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \overline{A}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \overline{A}_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The complete code follows in Listing 4.3.

Listing 4.3: *Fusion* implementation of model (4.4).

```
import sys
import mosek
from   mosek.fusion import *

def main(args):

    with Model("sdo1") as M:
        C  = Matrix.dense ( [[2.,1.,0.],[1.,2.,1.],[0.,1.,2.]] )
        A1 = Matrix.dense ( [[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]] )
        A2 = Matrix.dense ( [[1.,1.,1.],[1.,1.,1.],[1.,1.,1.]] )


        # Setting up the variables
        X = M.variable("X", Domain.inPSDCone(3))
        x = M.variable("x", Domain.inQCone(3))

        # Objective
        M.objective(ObjectiveSense.Minimize, Expr.add(Expr.dot(C, X), x.index(0)))

        # Constraints
        M.constraint("c1", Expr.add(Expr.dot(A1, X), x.index(0)), Domain.equalsTo(1.0))
        M.constraint("c2", Expr.add(Expr.dot(A2, X), Expr.sum(x.slice(1,3))), Domain.
↪equalsTo(0.5))

        M.setLogHandler(sys.stdout)

        M.solve()

        print(X.level())
        print(x.level())

if __name__ == '__main__':
    main(sys.argv[1:])
```

4.4 Integer Optimization

An optimization problem where one or more of the variables are constrained to integer values is denoted an integer optimization problem.

Section *4.4.2* shows *how to input an initial feasible solution* to help the solver.

4.4.1 Example MILO1

In this section the example

$$
\begin{array}{lccl}
\text{maximize} & x_0 + 0.64x_1 & & \\
\text{subject to} & 50x_0 + 31x_1 & \leq & 250, \\
 & 3x_0 - 2x_1 & \geq & -4, \\
 & x_0, x_1 \geq 0 & & \text{and integer}
\end{array}
\tag{4.5}
$$

is used to demonstrate how to solve a problem with integer variables.

The example (4.5) is almost identical to a linear optimization problem (see *4.1*) except for some variables being integer constrained. Therefore, only the specification of the integer constraints requires something new compared to the linear optimization problem discussed previously.

The complete source for the example is listed in Listing 4.4.

Listing 4.4: How to solve problem (4.5).

```
import sys

import mosek
import mosek.fusion
from   mosek.fusion import *

def main(args):
  A = [ [50.0, 31.0],
        [3.0, -2.0] ]
  c = [  1.0, 0.64]

  with Model('milo1') as M:

    x = M.variable('x', 2, Domain.integral(Domain.greaterThan(0.0)))


    # Create the constraints
    #      50.0 x[0] + 31.0 x[1] <= 250.0
    #       3.0 x[0] -  2.0 x[1] >= -4.0
    M.constraint('c1', Expr.dot(A[0], x), Domain.lessThan(250.0))
    M.constraint('c2', Expr.dot(A[1], x), Domain.greaterThan(-4.0))

    # Set max solution time
    M.setSolverParam('mioMaxTime', 60.0)
    # Set max relative gap (to its default value)
    M.setSolverParam('mioTolRelGap', 1e-4)
    # Set max absolute gap (to its default value)
    M.setSolverParam('mioTolAbsGap', 0.0)

    # Set the objective function to (c^T * x)
    M.objective('obj', ObjectiveSense.Maximize, Expr.dot(c, x))

# Solve the problem
    M.solve()

# Get the solution values
    ss = M.getPrimalSolutionStatus()
    print(ss)
    sol = x.level()
    print('[x0, x1] = ', sol)
    print("MIP rel gap = %.2f (%f)" % (M.getSolverDoubleInfo("mioObjRelGap"),M.
↪getSolverDoubleInfo("mioObjAbsGap")))

if __name__ == '__main__':
  main(sys.argv[1:])
```

4.4.2 Specifying an initial solution

Integer optimization problems are generally hard to solve, but the solution time can often be reduced by providing an initial solution for the solver. It is not necessary to specify the whole solution. By setting the *mioConstructSol* parameter to `on` and inputting values for the integer variables only, will force **MOSEK** to compute the remaining continuous variable values.

If the specified integer solution is infeasible or incomplete, **MOSEK** will simply ignore it.

Consider the problem

$$
\begin{array}{ll}
\text{maximize} & 7x_0 + 10x_1 + x_2 + 5x_3 \\
\text{subject to} & x_0 + x_1 + x_2 + x_3 \leq 2.5 \\
& x_0, x_1, x_2 \in \mathbb{Z} \\
x_0, x_1, x_2, x_3 \geq 0 &
\end{array}
\tag{4.6}
$$

The following example demonstrates how to optimize the problem using a feasible starting solution generated by selecting the integer values as $x = \{0, 2, 0, 1\}$ by the method *Variable.setLevel* .

The fundamental step is to feed **MOSEK** with the putative (feasible) solution: this is doe using the *Variable.setLevel* method, as reportedin Listing 4.5.

Listing 4.5: *Fusion* implementation of problem (4.6) specifying an initial solution.

```
x.setLevel(init_sol)
```

The complete code follows in Listing 4.6.

Listing 4.6: *Fusion* implementation of problem (4.6) specifying an initial solution.

```
import sys

import mosek
import mosek.fusion
from   mosek.fusion import *

def main(args):
  c = [ 7.0, 10.0, 1.0, 5.0 ]
  init_sol = [0.0, 2.0, 0.0, 1.0 ]

  with Model('mioinitsol') as M:

    n = 4

    x = M.variable('x', n, Domain.integral(Domain.greaterThan(0.0)))

    M.constraint( Expr.sum(x), Domain.lessThan(2.5))

    # Set max solution time
    M.setSolverParam('mioMaxTime', 60.0)
    # Set max relative gap (to its default value)
    M.setSolverParam('mioTolRelGap', 1e-4)
    # Set max absolute gap (to its default value)
    M.setSolverParam('mioTolAbsGap', 0.0)

    # Set the objective function to (c^T * x)
    M.objective('obj', ObjectiveSense.Maximize, Expr.dot(c, x))
    x.setLevel(init_sol)
    # Solve the problem
    M.solve()

    # Get the solution values
    ss = M.getPrimalSolutionStatus()
    print(ss)
    sol = x.level()
    print('x = ', sol)
    print("MIP rel gap = %.2f (%f)" % (M.getSolverDoubleInfo("mioObjRelGap"),M.
↪getSolverDoubleInfo("mioObjAbsGap")))

if __name__ == '__main__':
  main(sys.argv[1:])
```

CHAPTER

FIVE

DESIGN PRINCIPLES

Fusion has been designed based on many year of experience on Conic Optimization Problem. We believe that a dedicated API for conic optimization can be valuable to many **MOSEK** users that regularly solve Conic Optimization Problems and want to enjoy a simpler experience interfacing with the solver.

What *Fusion* is

An object-oriented framework for conic-optimization.

What *Fusion* is not

A modeling language.

Fusion is design for a fast and clean prototyping of Conic Optimization Problem, helping users in set and run problems quickly. At the same time users should not suffer excessive performance degradation.

Fusion has been design with the following ideas in mind:

- **Expressiveness**: we try to make it nice! Despite not being a modeling language, *Fusion* yields a pretty readable code that users mainteining and sharing their work.
- **Seamlessly multi-language** : *Fusion* users should be able to port *Fusion* based code across different supported languages with almost no modifications, except for those dependent on the languages themselves.
- **What you write is what ** |mosek| **gets**: *Fusion* do not perform any black-magic behind the scene! The model is feed into the solver with (almost) no additional transformations.

In the next section we elaborate on this topics.

5.1 A Seamless Multi-language API

Fusion has been designed to allow users to port their code easily across the supported programming languages. This means that all functionalities and naming conventions are the same, no matter what is the language.

The main purposes of this design choice are:

- *To make easier to share code among people using different languages* - In some settings people work together but like to use different languages.
- *To improve code reusability* - Code written in certain language may be needed in the future for other projects that use a different language.
- *To ease transition from R&D to production* - It is often easier to use fast-prototyping languages (as Python or MATLAB) during R&D, but production may required a different language, not least for performance sake.

As an example, let's see how a non-negative variable is declared in the supported languages:

Python

```
x = M.variable('x',1,Domain.greaterThan(0.))
```

Java

```
Variable x= M.variable("x", 1, Domain.greaterThan(0.));
```

C++

```
auto x= M->variable("x", 1, Domain::greaterThan(0.));
```

.NET

```
Variable x= M.Variable("x", 1, Domain.GreaterThan(0.0));
```

MATLAB

```
x= M.variable('x',1, Domain.greaterThan(0.));
```

The only significant differences are language related, i.e. they do not depends on *Fusion*. A careful coding can minimize such differences and improve even further the cross-language portability. Designing an interface spanning different programming languages is quite a challenge and leads to some limitations, as reported in Section *3.1.1* .

5.2 What You Write is What MOSEK Gets

Many object-oriented optimization frameworks allows a great flexibility of usage and they often support several solvers. The price to pay is that the model, as defined by the user, must be transformed in the formulation required by the solver. The framework may also perform some preprocessing autonomously (for instance scaling or conic reformulation). As result, the model formulated by the user may **not** be what the solver gets.

Fusion follows a different approach:

1. it clearly defines the formulation that the user must adhere to,
2. it only provides those functionalities required for that formulation,
3. it only performs transformations that involves additional variables required to correctly formulate conic constraints.

The only transformation that *Fusion* performs is the following: for each conic constraint of the form

$$Ax + b \in \mathcal{K} \tag{5.1}$$

with $A \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^n$, a vector of variables $y \in \mathbb{R}^m$ is introduced along with linear constraints, so that (5.1) is reformulated as

$$\begin{array}{c} y = Ax + b \\ y \in \mathcal{K} \end{array} \tag{5.2}$$

This mapping is necessary to ensure that each variable only belongs to one cone. Users accessing **MOSEK** through the low-level **MOSEK** Optimizer API for Python are required to make the mapping themselves. So *Fusion* does not make any transformation that the user would not have done himself.

Let's make an example: we want to define a constraint of the form

$$x_1 \geq \sqrt{(2x_2)^2 + (4x_3)^2}$$

which in conic form corresponds to

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \in Q \tag{5.3}$$

To bring equation (5.3) in the standard conic form accepted by the solver, we need to introduce additional variables y_i such that

1. each variable belongs to a single cone,
2. the cone can be formulated as $x_0 \geq \sqrt{\sum x_i^2}$.

Thus the user is forced to transform the constraint in the form

$$\begin{aligned} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} &= y, \\ y &\in Q \end{aligned} \tag{5.4}$$

The user can decide to make this kind of transformation himself, or let *Fusion* automatized the process. The results will be the same. To summarize:

- *Fusion* only allows user to define Conic Optimization Problem whose form is that of Problem (5.1).
- The only transformation performed by *Fusion* is the introduction of auxilary variables.
- Any other problem pre-solving and transformation are left to **MOSEK**.

The main benefits of this approach are:

1. The user knows what is the problem that the solver is actually solving.
2. Dual informations are readily available.
3. Reduced overhead.
4. Better control over numerical issues: scaling and other transformation may introduce numerical instability hard to detect.

CHAPTER

SIX

CONIC OPTIMIZATION MODELING

The main purpose of *Fusion* is to provide a simple and intuitive modeling API for conic linear optimization. A Conic Optimization Problem can be formulated compactly as

$$
\begin{array}{ll}
\text{minimize}_{x\in\mathbb{R}^n} & c^T x \\
s.t. & \\
 & A_i x + b_i \in \mathcal{K}_i \quad i = 1,\ldots,m
\end{array} \tag{6.1}
$$

where $\mathcal{K}_i \in \mathcal{D} = \{\mathbb{R}_+, \mathcal{Q}, \mathcal{Q}_r, \mathcal{S}_+\}$. *Fusion* extends $\mathcal{D}$ considering also some handy set defined by linear constraints:

- *ranged variable*: $\mathcal{R}^n = \{x \in \mathbb{R}^n | l \leq x \geq u\}$
- *upper bounded variable*: $\mathcal{U}^n = \{x \in \mathbb{R}^n | x \geq u\}$
- *lower bounded variable*: $\mathcal{L}^n = \{x \in \mathbb{R}^n | l \leq x\}$
- *unbounded variable*: $\mathbb{R}^n$

Then cone *Fusion* accepts cones such as $\mathcal{K}_i \in \mathcal{D} \cup \{\mathcal{R}, \mathcal{U}, \mathcal{L}, \mathbb{R}\}$.

The building blocks of a Conic Optimization Problem are

- *Variables*
- *Linear operators*
- *Domains*

Combining variables and linear operators we obtain affine functions that we can use to define the objective function and the constraint of our model.

- *Objective Function* we ask to minimize of maximize the affine function, see Section *6.7*.
- *Constraints* we ask that the image of affine function must belong to a given domain.

To create linear expression also matrices and vectors are needed. *Fusion* accepts plain arrays or matrices. However, it also provides simple classes to represent *dense and sparse matrices*.

Moreover, variables and epressions can be manipulated (stacking, reshaping or slicing) creating *logical views*.

Warning: A model built using *Fusion* is *always* a Conic Optimization Problem.

6.1 Optimization Model

The optimization model is the object that contains all information that define an conic optimization model:

$$
\begin{array}{ll}
\min & c^T x \\
s.t. & \\
 & Ax + b \in \mathcal{K}
\end{array}
$$

It is represented by the class *Model* and it is responsible for

- creating all the items define the optimization problem, i.e. variables, contraints and objective function;
- interface with the solver (see Section *8*).

This is particularly convenient because the user mainly interact with this class and, more importantly, leads to a safe and simple memory management. To create an optimization model, simply write

```
M = Model()
```

The name is optional. The returned object is what the user interacts with. It is important to keep in mind that

Important: A model owns all entities that it creates and it is responsible for their destruction.

As a consequence each model component can not be shared among models. There may be multiple models active at the same time.

Through an optimization model users can specify all the relevant component of an optimization model such

- *Variables*
- *Constraints*
- *Objective function*

All these elements must be created using the corresponding methods in *Model*, i.e. *Model.variable*, *Model.constraint* and *Model.objective*, respectively.

The *Model* is also the primary interface between the user and the solver (see Section *8*). For this reason it provides methods for

- set up parameters (see Section *8.1*)
- access problem and solution status (see Section *8.2*)
- perform I/O operations (see Section *8.3*)

Note: For those users familiar with the **MOSEK** Optimizer API: a *Model* instance is a wrapper on top of the problem `task`.

6.2 Matrices

Fusion provides a minimal support for matrix representation. The main purposes are

1. to provide the user with a convenient storage when the native language does not provide any,
2. to give the user the possibility to write a cross-platform code,
3. allow for a generic code in which sparse and dense matrices can be use with no modifications.

However, *Fusion* is **not** focused on providing matrix operations or any kind of linear algebra routines. The user should use specialized packages available for the programming language of interest.

Matrices in *Fusion* are stored either in *dense* or *sparse* format. Despite specific implementation are provided, the user is only supposed to interact with the generic interface *Matrix*. Matrices must be created by means of the static methods

- *Matrix.dense* for dense matrices,
- *Matrix.sparse* for sparse matrices.

Note: *Fusion* does not detect sparsity automatically.

A *Matrix* object is unmutable and therefore cannot be modified.

6.2.1 Dense Matrices

Dense matrices are the choice when the number of non zero entries is large. To specify a dense matrix one must use the *Matrix.dense* static method.

A dense matrix is specified providing its dimensions and the values that contains. It can not be left unspecified. Therefore the user must provide one of the following

- a common value for all entries: for instance

  $$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

  one may write the following code:

  ```
  ones= Matrix.dense(2,4,1.)
  ```

  A matrix with all entries equal to one can also be created by *Matrix.ones*.

- a complete set of values by a native representation, as for instance

  $$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

  one may write the following code:

  ```
  A= [ [1, 2, 3, 4], [5, 6, 7, 8] ]
  Ad= Matrix.dense(A)
  ```

- a flattened representation, i.e. all values stored in a one-dimensional array. For instance, the to declare

  $$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

  one may write the following code:

  ```
  A= [ 1, 2, 3, 4, 5, 6, 7, 8 ]
  Af= Matrix.dense(2, 4, A)
  ```

  The matrix is built row-wise.

6.2.2 Sparse Matrices

When the number of non zero elements is relatively small, then a sparse matrix is a preferable choice. It only stores the non zero elements in triplet form, i.e. each entry is represented by a triplet `(i,j,v)` where

- `i` is the row
- `j` is the column
- `v` is the non zero value

The order of the triplets is not relevant. To specify a sparse matrix one must use the *Matrix.sparse* static method.

For instance, the representation of

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

in triplet form is

$$A = \{(0,0,1),(0,3,1),(1,1,1),(1,3,1)\}$$

where we use the convention of 0-based indexes. In *Fusion* this corresponds to:

```
rows  = [ 0, 0, 1, 1 ]
cols  = [ 0, 3, 1, 2 ]
values= [ 1., 1., 1., 1. ]

print Matrix.sparse( len(rows), len(cols), rows, cols, values)
```

Runnig the code will result in the following output

```
SparseMatrix(2,4, (0,0,1.0),(0,3,1.0),(1,1,1.0),(1,2,1.0) )
```

Fusion provides also helper functions to create some of the most used sparse matrices:

- a diagonal matrix can be created simply by the *Matrix.diag* method;
- the identity matrix of size n can be created using the *Matrix.eye* method. It is a short-hand for the diagonal matrix.

6.2.3 Block Matrices

Many problems are characterized by linear expressions whose coefficient matrix has a block structure. *Fusion* allows to input block diagonal matrices, i.e.

$$M = \begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix},$$

where A, B and C may have different dimensions. Using the method *Matrix.diag* we can write

$$M = \begin{bmatrix} I_2 & 0 \\ 0 & 2I_3 \end{bmatrix},$$

with I_k being the identity matrix of dimension k, as

```
print( Matrix.diag([Matrix.eye(2), Matrix.diag(3,2.0)]).toString() )
```

The output is

```
'SparseMatrix(5,5, [(0,0,1.0),(1,1,1.0),(2,2,2.0),(3,3,2.0),(4,4,2.0) ])'
```

See *Matrix.diag* for details.

6.3 Domains

A domain specifies the set in which a linear expression must belong to. In particular, as explained in Section *6.5*, the feasible set of a conic optimization problem is defined by a set of constraints of the form:

$$A_i x + b_i \in \mathcal{K}_i, \quad i = 1, \ldots, m$$

for suitable matrices A_i and vectors b_i. Each constraint represents the intersection of a cone with a affine set generated by the linear expression. For sake of simplicity we will focus instead on the expression

$$y \in \mathcal{K}.$$

MOSEK solves conic optimization problems involving the fundamental symmetric cones:

1. *positive orthant:* $\mathbb{R}^n_+ = \{x \in \mathbb{R}^n | x \geq 0\}$
2. *Lorentz cone:* $\mathcal{Q}^{1+n} = \left\{(y,x) \in \mathbb{R}^{1+n} | y \geq \|x\|_2\right\}$
3. *Rotated cone:* $\mathcal{Q}^{2+n}_r = \left\{(y,w,x) \in \mathbb{R}^{1+n} | 2yw \geq \|x\|_2, w \geq 0, y \geq 0\right\}$
4. *PSD matrices:* $\mathcal{S}^n_+ = \left\{X \in \mathcal{S}^n | y^T X y \geq 0 \quad \forall y \in \mathbb{R}^n\right\}$

With these fundamental cones we can describe all constraints supported by **MOSEK**.

Domains are represented by specific classes. To declare a domain, we use the corresponding static method in *Domain*, as listed in the following table.

	Name	class method
Linear	unboundness	*Domain.unbounded*
	equality	*Domain.equalsTo*
	inequality $\leq$	*Domain.lessThan*
	inequality $\geq$	*Domain.greaterThan*
Quadratic	Lorentz cone	*Domain.inQCone*
	Rotated Lorentz cone	*Domain.inRotatedQCone*
Symm. Matr.	PSD matrix	*Domain.inPSDCone*
	linear PSD matrix	*Domain.isLinPSD*
	tri-linear PSD matrix	*Domain.isTrilPSD*

Warning: If no domain is specified for a variable, then the variable is left unconstrained!

6.3.1 Linear Domains

Linear domains are based on the positive orthant cone

$$\mathbb{R}_+ = \{x \in \mathbb{R} | x \geq 0\},$$

combined with an affine transformation $x = ay + b$. That allows to define other useful and commonly used domains:

- $x \geq b$ setting $a = 1$ (*Domain.greaterThan*),
- $x \leq b$ setting $a = -1$ (*Domain.lessThan*),

Fusion also contemplates explicitly unboundeness by *Domain.unbounded*.

6.3.2 Quadratic Cones

Both

- *Lorentz cone:* $\mathcal{Q}^{1+n} = \left\{(y,x) \in \mathbb{R}^{1+n} | y \geq \|x\|_2\right\}$ and
- *Rotated cone:* $\mathcal{Q}^{2+n}_r = \left\{(y,w,x) \in \mathbb{R}^{1+n} | 2yw \geq \|x\|_2, w \geq 0, y \geq 0\right\}$

are available using the *Domain.inQCone* and *Domain.inRotatedQCone* static methods, respectively. It must be understood that it is possible to express one cone in terms of the other bby means of an orthogonal transformation. That means one of the two cone is somehow redundant, but often very useful fro ma modeling perspective. For instance, if we want to express a contraint like

$$||x||_2 \leq k,$$

then the Lorentz cone exactly matches this constraint definition. But on the other hand,

$$x^T F F^T x = ||Fx||^2 \leq k$$

can be easily expressed by a rotated cone as

$$y = Fx, \left(\frac{1}{2}, k, y\right) \in \mathcal{Q}_r$$

Tip: Try to use the cone that closely match the problem definition!

6.3.3 Semidefinite Matrices

In *Fusion* there are three different domains derived from the cone of the semidefinite matrices.

Warning: None of the following domains explicitly enforce symmetry!

Symmetrized PSD

Given a matrix $X \in \mathbb{R}^{n \times n}$ the domain imposes the constraint

$$\frac{1}{2}(X + X^T) \in \mathcal{S}_+^n$$

This is available using the `Domain.inPSDCone`.

Lower Triangular PSD Domain

Given a matrix $X \in \mathbb{R}^{n \times n}$ the domain imposes the constraint

$$\begin{aligned} Y_{ij} = X_{ij} \quad i \geq j \\ Y \in \mathcal{S}_+^n \end{aligned}$$

This is available using the `Domain.isTrilPSD`.

Warning: The upper triangular part of X is left unspecified!

Linearized PSD Domain

6.3.4 Domain Size and Dimensions

Each domain has an intrinsic number of dimenions and minimum size, listed in table Table 6.3.4.

Domain	Dimensions	Minimum size
Linear	0	1
Lorentz cone	1	2
Rotated Cone	1	3
Symm. Matr.	2	1

The size and number of dimensions of a domain must match those of the object it must contains. However the size of a domain may or may not be fully specified: this gives *Fusion* the freedom to adapt the domain in order to match the dimension of the corresponding variable/constraint. If it is not possible an exception `FusionException` is thrown. See Section 6.5.

6.3.5 Integral Domains

For all domains, except those involving semidefinite matrices, the method *Domain.integral* can be used to restrict a given domain only to the integer values it contains. Therefore the specifier *Domain.integral* can only be used in combination to other domains. For instance, to declare a single integer variable $z \in [1, 10]$ we may write:

```
z= M.variable('z', Domain.integral(Domain.inRange(1.,10.)) )
```

Notice that binary variables are a special case of integer variables, and therefore to declare a binary variable `x` we may write

```
x= M.variable('x', Domain.integral( Domain.inRange(0.,1.) ) )
```

A handy specialized domain is provided by the *Domain.binary* function. For example, a binary variable $y \in \{0,1\}^n$ can be declared as

```
y= M.variable('y', Domain.binary())
```

Integrality can also be forced or relaxed after a variable has been created by means of the method *Variable.makeContinuous* and *Variable.makeInteger*.

6.4 Variables

In *Fusion* variables are objects that represent n-dimensional arrays. The base class *Variable* is the main interface the user works with.

Variables are declared using the *Model.variable* method that returns an object of type *Variable* representing the variable itself.

Important:

- a variable belongs to the model it is constructed by,
- the variable dimension and shape are immutable.

On the other hand, reshaped views can be easily obtained (see Section *6.8*).

The information that characterize a variable are:

- the variable shape and dimensions;
- the domain it must belong to using a domain specifier class of type *Domain* (see Section *6.3*).
- an optional name.

For instance, to declare a non-negative one-dimensional variable x of length n we may write

```
x = M.variable("x", n, Domain.greaterThan(0.))
```

A multidimensional array is declared simply specifying an array with all dimension sizes. For instance, a bi-dimensional $n \times n$ matrix of unbounded variables x can be declared as

```
x = M.variable( [n,n], Domain.unbounded() )
```

Many other combinations of parameters are available and allow to declare also semidefinte matrices and integer variables (see Section *6.4.1*). All variant return an implementation of the *Variable* base class. This type of object is a placeholder that can be used to

- form linear expressions and define constraints (see *6.5*),

- check the problem and solution status, along with the returned primal and dual values, after the optimization (see 8.2).

6.4.1 Integer Variables

Integer variables are expressed in the same way as the continuous, but with an additional domain specification to force integrality. *Fusion* will consider all integers in the specified domain. To add an integer variable $z \in [1, 10]$ we write

```
z= M.variable('z', Domain.integral(Domain.inRange(1.,10.)) )
```

Binary variables are declared either as

```
x= M.variable('x', Domain.integral( Domain.inRange(0.,1.) ) )
```

or with the helper function *Domain.binary*

```
y= M.variable('y', Domain.binary())
```

Warning: The interval limits must be real numbers.

In addition, integrality can be relaxed or enforced using the switch methods *Variable.makeContinuous* and *Variable.makeInteger*.

6.4.2 Views

It is often convenient to access a subset of a variable elements, to combine two or more variables or to reorganize a multidimensional variable in a different shape. For this reason *Fusion* provides a set of functions that return a *view* of the original variables..

In all cased the returned object still refers to the original one. It is just a logical placeholder. In the next subsections we will give some more details. For more details please refer to Section 6.8.

Reshaping

It keeps the same overall size but different number of dimension or their length. If the new shape is not compatible with the original size, an excpetion of type *DimensionError* is thrown. Available functions include

Function	Description
Var.reshape	General reshaping
Var.flatten	Returns a one dimensional representation
Var.compress	Remove all redundant dimensions of length one

Vector and matrix variables can also be transpose by *Variable.transpose*.

Slicing

It selects a subset of a variable element. Selection can be performed in different ways, the most common one being listed in the following table:

Function	Description
Variable.slice	select contiguous subsets
Variable.pick	select elements by set of indexes
Variable.index	select a single element by index

It is also possible to extract the diagonal of a two dimensional squared variable using the *Variable.diag* function.

Stacking

It returns a logical view that merges several variables in a single one. Stacking is useful to combine different variables in order to write a more compact set of contraints. We distinguish among three stacking operations, as in the following table

Function	Description
Var.hstack	concatenate along the first dimension
Var.vstack	concatenate along the second dimension
Var.stack	general block stacking

It is important to stress that the returned variable is only a *view* of the original ones. For more details, please refer to Section *6.8*.

6.4.3 Variable Naming

As stated in Section *6.4*, an optional name can be specified for each variable. This is particularly useful when debugging or storing a model, or reporting results. Names must be specified when variables are declared and cannot be changed afterwards. Variable indices are automatically generated.

When saving a model to file, *Fusion* will also include all the auxiliary variables that has been generated during the model building phase. The naming of these variables follows the following rules:

A careful choice of meaningful names can be of great help. *Fusion* puts no limitations on the names but the following rules apply:

1. names must be unique for the model the variable belongs to;
2. the value *None is* allowed and correspond to automatic names;
3. they must not collide with automatically generated names.

6.4.4 Pretty Printing

Variable information can be printed out in a human-readable form using the method *Variable.toString*. The text contains

- type,
- name and
- dimension.

The textual representation is generated as compact as possible. For instance a one dimensional variable called x will be printed just saying

```
n = 4
x = M.variable("x", n, Domain.greaterThan(0.))
print( x.toString() )
```

with the following output

```
LinearVariable( ('x',4) )
```

Notice that if no names are assigned an empty string will be used instead, i.e. no automatic names are generated.

6.5 Linear Expressions

In *Fusion* linear expressions are constructed combining *variables* and *matrices* by linear operators. The result is an object that represent the linear expression itself.

Important: *Fusion* only allows for those combinations of operators and arguments that yields linear functions.

For instance the dot product between two vector of variables is not allowed, as it yields a quadratic function. As a consequence, *at most one of the arguments of the expression can be a variable.*

For a given expression, we define as

- **dimension** the dimension of the result of the expression,
- **size** the product of the expression *dimension* times the input dimension.

For instance, given $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$, the dimension of Ax is m and its size is nm.

Linear expressions are used to define the *constraints* and the *objective function*.

6.5.1 Storing Expressions

Expressions are concrete implementations of the virtual interface *Expression*.

Note: Typically one never needs to directly use *Expression* and its descendants.

Expressions are organized in matrices, and therefore they inherit the possibility to be sliced, reshaped and stacked. These operations could be useful in practice and yield very compact formulations. For instance, if we want to express

$$Ax_i = b_i \quad i = 1, \ldots, n$$

we can think of $x_i \in \mathbb{R}^m$ as the column of a matrix and therefore write simply

$$AX = B, \quad X = [x_1 \ldots x_n], \quad B = [b_1 \ldots b_n].$$

The resulting expression AX has dimension $n \times m$, i.e. is a matrix expression.

6.5.2 Defining Expressions

Linear operators are provided as static method by the *Expr* class. Each operator returns an expression object of type *Expression*.

Fusion currently support the linear operators listed in Table 6.1.

Table 6.1: Linear Operators

Method	Description
Expr.add	Element-wise addition of two matrices
Expr.sub	Element-wise subtraction of two matrices
Expr.mul	Matrix multiplication
Expr.neg	Sign inversion
Expr.outer	Vector outer-product
Expr.dot	Dot product
Expr.sum	Sum over a given dimension
Expr.mulDiag	Sum over the diagonal of a matrix which is the result of a matrix multiplication
Expr.constTerm	Return a *constant term*

Note that some of the operators are provided for user convinence, as they could be obtained by means of others. Please click on te corresponding link to see more details.

Dimensionality checking

Fusion perform dimensionality checking on the arguments of a linear operators: an exception of type *DimensionError* will be thrown if errors are detected.

Composing expressions

Expression can be composed and nested. This allow to define more complex expressions: for instance say we want to write

$$Ax + By,$$

for appropriate matrices A, B and variables x, y. The code will look like:

```
Expr.add( Expr.mul(A,x), Expr.mul(B,y) )
```

Given that expressions can be stored, one can also define expression separately and then combine them:

```
Ax = Expr.mul(A,x)
By = Expr.mul(B,y)

Expr.add(Ax,By)
```

Composition is pretty useful, but it may lead to unreadable code. Users should also consider using list-based expressions if possible. To write an expression such as

$$x + y + z + w$$

where x, y, z, w are all vectors of variables of the same size, a first option is

```
Expr.add(x, Expr.add(y, Expr.add(z,w) ) )
```

which is not very much readable. A cleaner way can be to store all terms in a list, for instance

```
Expr.add( [x, y, z, w] )
```

Similar function are provided for other expressions.

6.6 Constraints

A constraint in *Fusion* must have the form

$$l(x) \in F$$

where $l(x)$ is an affine function (see section *6.5*), and F must be a domain among those provided by *Fusion* (see Section *6.3*).

A constraint is characterized by its

- *type*
- *size*
- *number of dimensions* of the image of $l(x)$,

- *number of non-zeros entries* are the actual number of terms that defines the affine function. This is one of the most important measure of the actual dimension of a Conic Optimization Problem.

For instance, the following set of linear constraints

$$\begin{array}{ccccccc} x_1 & + & 2x_2 & & & = 0 \\ & + & x_2 & + & x_3 & = 0 \\ x_1 & & & & & = 0 \end{array} \tag{6.2}$$

has size three, number of dimensions equals to one (the image is indeed a one dimensional array) and five non zero elements.

The dimensions of F and $l(x)$ must match, otherwise an error is reported. To this end *Fusion* tries to smartly deduce the dimension of F, whenever possible and safe, to that of $l(x)$.

6.6.1 Contraint Declaration

Constraints must be created using the static method *Model.constraint*. A constraint is defined by three parameters:

1. *An optional name*: it is useful for debugging purposes or when dumping the model to file. However, it may introduce a significant overhead for large models.
2. A linear expression
3. The domain of the image of the linear expression

The *Model.constraint* method returns a *Constraint* object that can be used by the user to access constraint information, see Section *Constraint Information*.

For instance, the set of linear constraints (6.2) can be declared as

```
A = [ [1.0, 2.0, 0.0],
      [0.0, 1.0, 1.0],
      [ 1.0, 0.0, 0.0] ]

x = M.variable("x",3,Domain.unbounded())
c = M.constraint( Expr.mul(A,x), Domain.equalsTo(0.0))
```

6.6.2 Constraint Information

There are several information that can be obtained from a costraint object.

The *size* and number of *number of dimensions* are available using the methods *Constraint.size* and *Constraint.get_nd*.

It is also possible to recover dual information using the method *Constraint.dual*.

Warning: Dual information are only available for continuous problems and if a (near) optimal solution has been found!

6.6.3 Pretty Printing

A human readable representation of the constraint can be obtain using the *Constraint.toString* method.

The representation of a *Constraint* instance is the list of all contained constraints. For instance a set of linear constraints of the form $Ix = 0$, with I being the identity matrix is implemented as

```
n = 4
x = M.variable('x',n, Domain.greaterThan(0.))
c = M.constraint('c', Expr.mul(Matrix.eye(n),x), Domain.equalsTo(0.))

print(c.toString())
```

The output is

```
Constraint( 'c', (4),
  c[0] :  + 1.0 x[0]  = 0.0,
  c[1] :  + 1.0 x[1]  = 0.0,
  c[2] :  + 1.0 x[2]  = 0.0,
  c[3] :  + 1.0 x[3]  = 0.0 )
```

Notice that only non zero entries are printed.

The printed representation also includes all auxiliary variables introduced by *Fusion*. For instance a single second order cone of the form

$$(t, x) \in \mathcal{Q}$$

with $t \in \mathbb{R}, x \in \mathbb{R}^n$, implemented as

```
n = 4

x = M.variable("x", n, Domain.greaterThan(0.))
t = M.variable("t", 1, Domain.greaterThan(0.))

c = M.constraint( Expr.vstack(t,x), Domain.inQCone())

print(c.toString() )
```

it will produce

```
ConicConstraint( (5), QuadCone,
 c[0] :  + 1.0 t_[0]  : element in a quadratic cone,
 c[1] :  + 1.0 x_[1]  : element in a quadratic cone,
 c[2] :  + 1.0 x_[2]  : element in a quadratic cone,
 c[3] :  + 1.0 x_[3]  : element in a quadratic cone,
 c[4] :  + 1.0 x_[0]  : element in a quadratic cone)
```

This method is particulary useful when the model must be inspected for debugging. However it must be notice that

- when the number of variable involved in the constraint is large, it may generate a large amount of output as well;
- meaningful names must be provided for the relevant variables.

Warning: If no names are given, *Fusion* will just display empty strings!

6.7 Objective Function

In *Fusion* the objective function must be an affine function of size one, i.e. returning a scalar, otherwise an exception of type *DimensionError* is thrown.

The optimization sense can be either *minimize* or *maximize* and

Note: in *Fusion* the optimization sense must always be specified.

The only exception is the trivial case in which the objective function is a constant term.

The objective function is declared using the static method *Model.objective*. It requires:

- *the optimization sense* from the enumeration *ObjectiveSense*,
- *the linear expression* defining the objective function and an optional name, see Section *6.5*.

For instance, to minimize a variable t we will write

```
with Model() as M:

    t = M.variable('t', 1, Domain.unbounded() )

    M.objective( ObjectiveSense.Minimize, t)
```

The typical linear objective function $c^T x$ can be declared as

```
c = [1.0, 1.0, 1.0]
n = len(c)
x = M.variable("x", n, Domain.greaterThan(0.0))
M.objective( ObjectiveSense.Minimize,  Expr.mul(c,x))
```

Note that the objective function is a little peculiar in *Fusion*:

- it is the *only* component of an optimization model that can not be stored and reused,
- it cannot be modified,
- it can be overwritten.

6.7.1 Changing the Objective Function

The objective function can be overwritten at any time. This is particularly useful when solving a sequence of problems in which only the objective function varies. For instance, if we want to minimize the following linear function

$$f(x) = \gamma x + \beta y,$$

where $\gamma, \beta, x, y \in \mathbb{R}$, for different values of the parameter $\gamma > 0$. The function is trivial, but it conveys the overall ideas. We may use the following code:

```
gamma=[0., 0.5, 1.0]
beta = 2.0
with Model() as M:
    x = M.variable('x',1,Domain.greaterThan(0.))
    y = M.variable('y',1,Domain.greaterThan(0.))
    beta_y = Expr.mul(beta,y)
    for g in gamma:

      M.objective( ObjectiveSense.Minimize,Expr.add( Expr.mul(g,x) , beta_y) )
      M.solve()
```

This is particularly useful when performing for instance multi/objective optimization.

Tip: Notice how the common expression can be stored and reused.

6.8 Variable and Expression Views

In *Fusion* variables and expressions are organized as multi-dimensional objects. We will refer in general to *matrix-like object*, or more simply to a *matrix* meaning both variables and expressions.

Matrix-like objects are characterized by a **shape** that accounts for the dimension of the space in which the object lives in. The shape is represented by an array of integers such that

- the lenght is the **dimension** on the matrix,
- each entry is the size along that dimension and
- the product of all dimensions is the **size** of the matrix.

For instance the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

has shape $\{2, 3\}$, dimension 2 and size $2 \times 3 = 6$.

The shape is an intrinsic property that is **immutable** in *Fusion*, and can be obtain by the methods `Variable.getShape` and `Expression.getShape`.

It is often useful to

- re-organize matrix elements in a different way, i.e. *reshaping*,
- only consider a subset of elements, i.e. *picking and slicing* and
- pack matrices together, i.e *stacking*.

All these operations do not require *new* variables or expressions, but just logical **views**. *Fusion* provides an unified set of methods to creates views for variables and expressions. It is must be stressed that

Important: A views does not introduce neither new variables nor additional constraints. In general it is a lightweight object that only requires a small amount of memory.

6.8.1 Reshaping

Reshaping a matrix means to rearrange its elements in a different shape, with the same size.

Flattening

This operation returns a **one-dimensional** representation of the matrix. The elements are listed traversing the matrix in row-wise order. For a two dimensional matrix $x \in \mathbb{R}^{n \times m}$ the result of the flatting operation is

$$[x_{11}, x_{12}, ..., x_{1m}, ..., x_{n1}, x_{n2}, ..., x_{nm}]$$

For instance the flattening of

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{6.3}$$

is the following one-dimensional array f

$$f^T = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6]^T. \tag{6.4}$$

Flattening is available both for variables (see `Var.flatten`) and expressions (see `Expr.flatten`).

General Reshaping

It returns a view of the matrix with a different shape with the same size. The entries of the original matrix are mapped as follows:

1. first the matrix is *flattened* yielding an array f,
2. a new matrix with the new shape is created,
3. the new matrix is filled using f.

Note that in general the number of dimensions may differs. The only strict requirements is that the size must match.

For instances, let's assume we are given a matrix as follows

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \tag{6.5}$$

but we need to see it as a matrix with three rows and two columns. A is first flattened, as in (6.4) and then its values rolled over the new shape. The final result is

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}. \tag{6.6}$$

Flattening is available both for variables (see *Var.reshape*) and expresisons (see *Expr.reshape*).

6.8.2 Slicing and Picking

Sometimes constraints and objective function only involve a sub-set of variables. It is then useful to have a way to select them in a compact way.

- **picking**: it selesct a subset of possibly non-contiguous variable entries.
- **slicing**: it selects a continuous sebset of variable entries.

Clearly *slicing* is a special case of *picking*. However, *slicing* occours so frequently that deserve dedicated methods.

Picking

Picking is the operation of selecting elements based on a list of indexes. The resulting view is a one dimensional array.

For a given variable x, in the general case the user must provide a list L of indexes to identify the items and the resulting array p will be arranged such that:

```
p[i] = x[L[i]]
```

For instance, given a two dimensional matrix A of dimension $n \times n$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

to select the upper-left and bottom-right item we specify a list that contains the coordinates of the elements, i.e

$$\{(0,0),(1,2)\}.$$

The result is the one-dimeensional array $\{1,6\}$. If we specify

$$\{(1,2),(0,0)\}.$$

we obtain $\{6, 1\}$.

Note how the element coordinates are tuples with the same dimension as the matrix.

Fusion provides several variant of the *Variable.pick* and *Expression.pick* methods. Single element access by the methods *Variable.index* and *Expr.index* is a special cases of picking.

Slicing

Slicing refers to the selection of a submatrix. A slice is defined by the range of indexes selected for each dimension: to select the elements in the range *[first,last]*, we actually specify *[first,last+1] = [l_i, u_i]*. In this way the length of the slice along that dimension is exactly *[u_i-l_i]*.

Consider a matrix with shape 4×4, as depicted in Fig. 6.1.

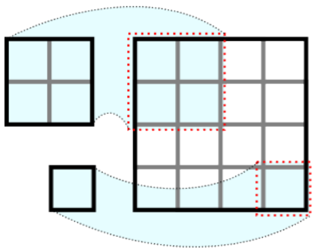


Fig. 6.1: Two dimensional slicing.

To select the upper-left 2×2 sub-matrix, we specify the range *[0,2]* for both dimensions; on the other hand, to select the cell on the bottom right corner, we use `[3,4]` again for both dimensions.

In *Fusion* slicing is obtained using the methods *Variable.slice* and *Expression.slice*, providing separate arrays for the starting and ending indexes. For instance, the previous example would require

- a first array `[0,0]` and a second `[2,2]` for the upper left corner,
- a first array `[3,3]` and a second `[4,4]` for the lower right corner.

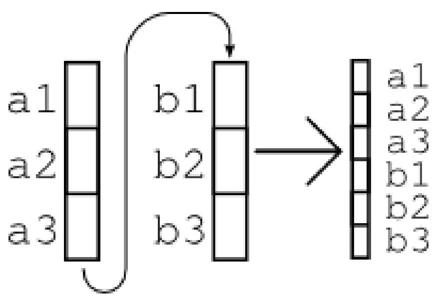
Fusion allows for slices in which dimensions can have zero length, i.e. where *last* equals to *first*. If *last* is less than *first*, an exception of type *IndexError* is thrown.

6.8.3 Stacking

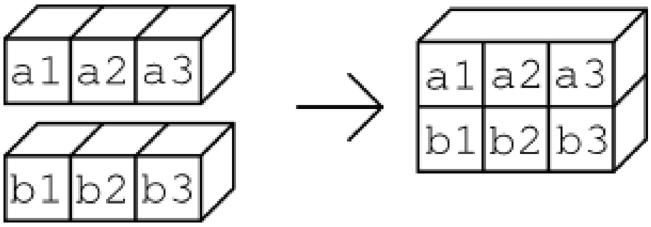
Stacking refers to the concatenation of matrices to form a new larger one.

Vertical Stacking

It concatenates matrices along the first dimension. In case of bi-dimensional matrices, they are put one on top of the other, i.e. vertically. For instance, given two column vectors a, b, i.e. with second dimension equals to one, the vertical stacking of a on top of b is depicted in the following figure.



On the other hand, if the second dimension is one, the results is the following



In *Fusion* this operation is performed by the *Expr.vstack* and *Var.vstack* functions.

Horizontal stacking

It concatenates matrices along the second dimension. In case of bi-dimensional matrices, they are put one beside the other, i.e. horizontally.

In *Fusion* this operation is performed by the *Expr.hstack* and *Var.hstack* functions.

Generalized stacking

It allows to combine several matrices as long as their dimensions match. For instance consider Fig. 6.2: five two-dimensional matrices must be combine to obtain a larger one.

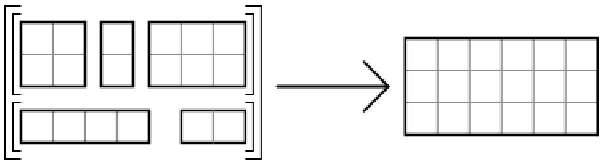
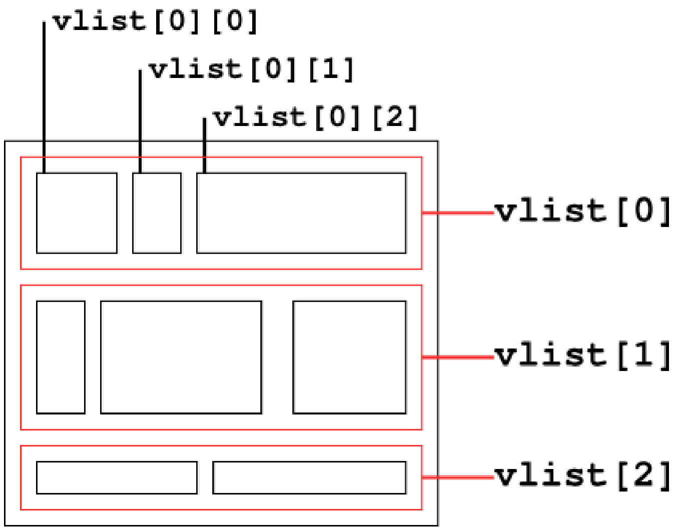


Fig. 6.2: An example of general stacking.

General stacking is supported for both variable (*Var.stack*) and expressions (*Expr.stack*).

> **Warning:** Variables and expressions cannot mixed when stacking! You must promote variables to expressions using *Variable.asExpr*.

To better explain how stack works let's consider the case in Fig. 6.8.3



The matrices are stored in a two-dimensional array named `vlist` such that

- Each rows of `vlist` contains matrices with the same number of rows and
- Each rows has the same total number of columns.

If `vlist` is composed by variables, then we may write something along this line

```
vlist = [
    [M.variable([2,2]), M.variable([2,1]), M.variable([2,6])],
    [M.variable([3,1]), M.variable([3,5]), M.variable([3,3])],
    [M.variable([1,4]), M.variable([1,5])                   ]
]

M.constraint( Var.stack(vlist), Domain.equalsTo(0.))
```

CHAPTER

SEVEN

CASE STUDIES

In this section we present some case studies in which the Fusion API for Python is used to solve real-life applications. These examples involve some more advanced modeling skills and possibly some input data. The user is strongly recommended to first read the *basic tutorials* before going through these advanced case studies.

Case Studies	Type	Int.	Keywords
Portfolio Optimization	CQO	NO	stacking, objective function change,
Primal SVM	CQO	NO	variable repeat
2D Total Variation	CQO	NO	slicing, sliding windows
Inner and outer Löwner_John Ellipsoids	SDO	NO	determinant root
Nearest Correlation Matrix Problem	SDO	NO	nuclear norm
Semmidefinite relaxation of MIQCQP problems	SDO	NO	
SUDOKU Game	MILP	YES	assignement constraints
Multi_Processors Scheduling	MILP	YES	assignement constraints, initial solution
Travelling Sales_Man	MILP	YES	graph, row generation

7.1 Portfolio Optimization

This case studies is devoted to the Portfolio Optimization Problem.

7.1.1 The Basic Model

The classical Markowitz portfolio optimization problem considers investing in n stocks or assets held over a period of time. Let x_j denote the amount invested in asset j, and assume a stochastic model where the return of the assets is a random variable r with known mean

$$\mu = \mathbf{E}r$$

and covariance

$$\Sigma = \mathbf{E}(r-\mu)(r-\mu)^T.$$

The return of the investment is also a random variable $y = r^T x$ with mean (or expected return)

$$\mathbf{E}y = \mu^T x$$

and variance (or risk)

$$\mathbf{E}(y - \mathbf{E}y)^2 = x^T \Sigma x.$$

The problem facing the investor is to rebalance the portfolio to achieve a good compromise between risk and expected return, e.g., maximize the expected return subject to a budget constraint and an upper bound (denoted γ) on the tolerable risk. This leads to the optimization problem

$$\begin{array}{lccl} \text{maximize} & \mu^T x & & \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & x^T \Sigma x & \leq & \gamma^2, \\ & x & \geq 0. & \end{array} \tag{7.1}$$

The variables x denotes the investment i.e. x_j is the amount invested in asset j and x_j^0 is the initial holding of asset j. Finally, w is the initial amount of cash available.

A popular choice is $x^0 = 0$ and $w = 1$ because then x_j may be interpretated as the relative amount of the total portfolio that is invested in asset j.

Since e is the vector of all ones then

$$e^T x = \sum_{j=1}^{n} x_j$$

is the total investment. Clearly, the total amount invested must be equal to the initial wealth, which is

$$w + e^T x^0.$$

This leads to the first constraint

$$e^T x = w + e^T x^0.$$

The second constraint

$$x^T \Sigma x \leq \gamma^2$$

ensures that the variance, or the risk, is bounded by the parameter γ^2. Therefore, γ specifies an upper bound of the standard deviation the investor is willing to undertake. Finally, the constraint

$$x_j \geq 0$$

excludes the possibility of short-selling. This constraint can of course be excluded if short-selling is allowed.

The covariance matrix Σ is positive semidefinite by definition and therefore there exist a matrix G such that

$$\Sigma = GG^T. \tag{7.2}$$

In general the choice of G is **not** unique and one possible choice of G is the Cholesky factorization of Σ. However, in many cases another choice is better for efficiency reasons as discussed in Section *7.1.2*. For a given G we have that

$$\begin{array}{rcl} x^T \Sigma x & = & x^T G G^T x \\ & = & \left\|G^T x\right\|^2. \end{array}$$

Hence, we may write the risk constraint as

$$\gamma \geq \left\|G^T x\right\|$$

or equivalently

$$[\gamma; G^T x] \in Q^{n+1}.$$

where Q^{n+1} is the $n+1$ dimensional quadratic cone. Therefore, problem (7.1) can be written as

$$\begin{array}{lllll} \text{maximize} & \mu^T x & & & \\ \text{subject to} & e^T x & = & w + e^T x^0, & \\ & [\gamma; G^T x] & \in & Q^{n+1}, & \\ & x & \geq & 0, & \end{array} \tag{7.3}$$

which is a conic quadratic optimization problem that can easily be formulated and solved with *Fusion*. Subsequently we will use the example data

$$\mu = \left[\begin{array}{c} 0.1073 \\ 0.0737 \\ 0.0627 \end{array} \right]$$

and

$$\Sigma = 0.1, \left[\begin{array}{ccc} 0.2778 & 0.0387 & 0.0021 \\ 0.0387 & 0.1112 & -0.0020 \\ 0.0021 & -0.0020 & 0.0115 \end{array} \right].$$

This implies

$$G^T = \sqrt{0.1} \left[\begin{array}{ccc} 0.5271 & 0.0734 & 0.0040 \\ 0 & 0.3253 & -0.0070 \\ 0 & 0 & 0.1069 \end{array} \right]$$

We will make use of a simple helper function to compute the inner product of two vectors

```
# Computes the inner product between two vectors.
def dot(x,y):
    return sum([ xx*yy for xx,yy in zip(x,y) ]) + 0.
```

Listing 7.1 demonstrates how the basic Markowitz model (7.3) is implemented using *Fusion*.

Listing 7.1: Code implementing problem (7.3).

```
def BasicMarkowitz(n,mu,GT,x0,w,gamma):
    """
    Purpose:
        Computes the optimal portfolio for a given risk

    Input:
        n: Number of assets
        mu: An n dimensional vector of expected returns
        GT: A matrix with n columns so (GT')*GT  = covariance matrix
        x0: Initial holdings
        w: Initial cash holding
        gamma: Maximum risk (=std. dev) accepted

    Output:
        Optimal expected return and the optimal portfolio
    """

    with  Model("Basic Markowitz") as M:

        # Redirect log output from the solver to stdout for debugging.
        # if uncommented.
        # M.setLogHandler(sys.stdout)
```

```
        # Defines the variables (holdings). Shortselling is not allowed.
        x = M.variable("x", n, Domain.greaterThan(0.0))

        #  Maximize expected return
        M.objective('obj', ObjectiveSense.Maximize, Expr.dot(mu,x))

        # The amount invested  must be identical to initial wealth
        M.constraint('budget', Expr.sum(x), Domain.equalsTo(w+sum(x0)))

        # Imposes a bound on the risk
        M.constraint('risk', Expr.vstack( gamma,Expr.mul(GT,x)), Domain.inQCone())

        # Solves the model.
        M.solve()

        return dot(mu,x.level())
```

The source code should be self-explanatory except perhaps for

```
M.constraint('risk', Expr.vstack( gamma,Expr.mul(GT,x)), Domain.inQCone())
```

where the linear expression

$$\left[\begin{array}{c} \gamma G^T x \end{array} \right]$$

is created using the *Expr.vstack* operator. Finally, the linear expression must lie in a quadratic cone implying

$$\gamma \geq \left\| G^T x \right\|.$$

7.1.2 The Efficient Frontier

The portfolio computed by the Markowitz model is efficient in the sense that there no other other portfolio giving a strictly higher return for the same amount of risk. An efficient portfolio is also sometimes called a Pareto optimal portfolio. Clearly, an investor should only invest in efficient portfolios and therefore it may be relevant to present the investor for all efficient portfolios so the investor can choose the portfolio that has the desired tradeoff between return and risk.

Given a nonnegative α then the problem

$$\begin{array}{lll} \text{maximize} & \mu^T x - \alpha s & \\ \text{subject to} & e^T x & = w + e^T x^0, \\ & [s; G^T x] & \in Q^{n+1}, \\ & x & \geq 0. \end{array} \tag{7.4}$$

computes an efficient portfolio. Note that the objective maximize the expected return while maximizing $-\alpha$ times the standard deviation. Hence, the standard deviation is minimized while α specifies the tradeoff between expected return and risk. Ideally the problem (7.4) should be solved for all values $\alpha \geq 0$ but in practice impossible. Using the example data from Section 7.1.1, the optimal values of return and risk for several αs are listed below:

```
Efficient frontier

alpha          return         risk
0.0000         1.0730e-01     7.2700e-01
0.0100         1.0730e-01     1.6667e-01
0.1000         1.0730e-01     1.6667e-01
0.2500         1.0321e-01     1.4974e-01
0.3000         8.0529e-02     6.8144e-02
0.3500         7.4290e-02     4.8585e-02
```

```
0.4000        7.1958e-02    4.2309e-02
0.4500        7.0638e-02    3.9185e-02
0.5000        6.9759e-02    3.7327e-02
0.7500        6.7672e-02    3.3816e-02
1.0000        6.6805e-02    3.2802e-02
1.5000        6.6001e-02    3.2130e-02
2.0000        6.5619e-02    3.1907e-02
3.0000        6.5236e-02    3.1747e-02
10.0000       6.4712e-02    3.1633e-02
```

Example code

Listing 7.2 demonstrates how to compute the efficient portfolios for several values of α in *Fusion*.

Listing 7.2: Code for the computation of the efficient frontier based on problem (7.4).

```
def EfficientFrontier(n,mu,GT,x0,w,alphas):
    """
    Purpose:
        Computes several portfolios on the optimal portfolios by

            for alpha in alphas:
                maximize   expected return - alpha * standard deviation
                subject to the constraints

    Input:
        n: Number of assets
        mu: An n dimensional vector of expected returns
        GT: A matrix with n columns so (GT')*GT  = covariance matrix
        x0: Initial holdings
        w: Initial cash holding
        alphas: List of the alphas

    Output:
        The efficient frontier as list of tuples (alpha,expected return,risk)
    """

    with Model("Efficient frontier") as M:

        # M.setLogHandler(sys.stdout)

        # Defines the variables (holdings). Shortselling is not allowed.
        x = M.variable("x", n, Domain.greaterThan(0.0)) # Portfolio variables
        s = M.variable("s", 1, Domain.unbounded()) # Risk variable

        M.constraint('budget', Expr.sum(x), Domain.equalsTo(w+sum(x0)))

        # Computes the risk
        M.constraint('risk', Expr.vstack(s,Expr.mul(GT,x)),Domain.inQCone())

        frontier = []

        mudotx = Expr.dot(mu,x)

        for i,alpha in enumerate(alphas):

            #  Define objective as a weighted combination of return and risk
            M.objective('obj', ObjectiveSense.Maximize, Expr.sub(mudotx,Expr.mul(alpha,s)))

            M.solve()
```

```
        frontier.append((alpha,dot(mu,x.level()),s.level()[0]))

    return frontier
```

Note the efficient frontier could also have been computed using the code in Section *7.1.1* by varying γ. However, when the constraints of a *Fusion* model is changed the model has to be rebuild whereas a rebuild is not needed if only the objective is modified.

7.1.3 Improving the Computational Efficiency

In practice it is often important to solve the portfolio problem in a short amount of time. Therefore, in this section it is discussed what can be done at the modelling stage to improve the computational efficiency.

The computational cost is of course to some extent dependent on the number of constraints and variables in the optimization problem. However, in practice a more important factor is the number nonzeros used to represent the problem. Indeed it is often better to focus at the number of nonzeros in G see (7.2) and try to reduce that number by for instance changing the choice of G.

In other words if the computational efficiency should be improved then it is always good idea to start with focusing at the covariance matrix. As an example assume that

$$\Sigma = D + VV^T$$

where D is a positive definite diagonal matrix. Moreover, V is a matrix with n rows and p columns. Such a model for the covariance matrix is called a factor model index{factor model} and usually p is much smaller than n. In practice p tends be a small number independent of n say less than 100.

One possible choice for G is the Cholesky factorization of Σ which requires storage proportional to $n(n+1)/2$. However, another choice is

$$G^T = \left[\begin{array}{c} D^{1/2}V^T \end{array} \right]$$

because then

$$GG^T = D + VV^T.$$

This choice requires storage proportional to $n + pn$ which is much less than for the Cholesky choice of G. Indeed assuming p is a constant then the difference in storage requirements is a factor of n.

The example above exploits the so-called factor structure and demonstrates that an alternative choice of G may lead to a significant reduction in the amount of storage used to represent the problem. This will in most cases also lead to a significant reduction in the solution time.

The lesson to be learned is that it is important to investigate how the covariance is formed. Given this knowledge it might be possible to make a special choice for G that helps reducing the storage requirements and enhance the computational efficiency.

7.1.4 Slippage Cost

The basic Markowitz model assumes that there are no costs associated with trading the assets and that the returns of the assets is independent of the amount traded. None of those assumptions are usually valid in practice. Therefore, a more realistic model is

$$\begin{array}{lrcl} \mbox{maximize} & \mu^T x & & \\ \mbox{subject to} & e^T x + \sum_{j=1}^n T_j(x_j - x_j^0) & = & w + e^T x^0, \\ & x^T \Sigma x & \leq & \gamma^2, \\ & x & \geq & 0, \end{array} \tag{7.5}$$

where the function

$$T_j(x_j - x_j^0)$$

specifies the transaction costs when the holding of asset j is changed from its initial value.

7.1.5 Market Impact Costs

If the initial wealth is fairly small and no short selling is allowed, then the holdings will be small and then the amount traded of each asset must also be small. Therefore, it is reasonable to assume that the prices of the assets is independent of the amount traded. However, if a large volume of an assert is sold or purchased it can be expected that the price change and hence the expected return also change. This effect is called market impact costs. It is common to assume that the market impact cost for asset j can be modelled by

$$m_j\sqrt{|x_j - x_j^0|}$$

according where m_j is a constant that is estimated in some way. See *[GK00]* [p. 452] for details. Hence, we have

$$T_j(x_j - x_j^0) = m_j|x_j - x_j^0|\sqrt{|x_j - x_j^0|} = m_j|x_j - x_j^0|^{3/2}.$$

From *[MOSEKApS12]* it is known

$$\{(t,z) : t \geq z^{3/2}, z \geq 0\} = \{(t,z) : (s,t,z),(z,1/8,s) \in Q_r^3\}$$

where Q_r^3 is the 3 dimensional rotated quadratic cone. Hence, it follows

$$\begin{array}{ll} z_j & = |x_j - x_j^0|, \\ (s_j,t_j,z_j),(z_j,1/8,s_j) & \in Q_r^3, \\ \sum_{j=1}^n T(x_j - x_j^0) & = \sum_{j=1}^n t_j. \end{array}$$

Unfortunately this set of constraints is nonconvex due to the constraint

$$z_j = |x_j - x_j^0| \tag{7.6}$$

but in many cases the constraint may be replaced by the relaxed constraint

$$z_j \geq |x_j - x_j^0|. \tag{7.7}$$

which is equivalent to

$$\begin{array}{l} z_j \geq x_j - x_j^0, \\ z_j \geq -(x_j - x_j^0). \end{array} \tag{7.8}$$

For instance if the universe of assets contains a risk free asset then

$$z_j > |x_j - x_j^0| \tag{7.9}$$

cannot hold for an optimal solution.

Now given that the optimal solution has the property that (7.9) holds then the market impact costs within the model is larger than the true market impact cost and hence money are essentially considered garbage and removed by generating transaction costs. This may happen if a portfolio with very small risk is requested because then the only way to obtain a small risk is to get rid of some of the assets by generating transaction costs. It is assumed this is not the case and hence the models (7.6) and (7.7) are equivalent.

The above observations leads to

$$
\begin{array}{lllll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + m^T t & = & w + e^T x^0, & \\
 & (\gamma, G^T x) & \in & Q^{n+1}, & \\
 & z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
 & z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
 & [v_j; t_j; z_j], [z_j; 1/8; v_j] & \in & Q_r^3, & j = 1, \ldots, n, \\
 & x & \geq & 0. &
\end{array}
\tag{7.10}
$$

The revised budget constraint

$$e^T x = w + e^T x^0 - m^T t$$

specifies that the total investment must be equal to the initial wealth minus the transaction costs. Moreover, observe the variables v and z are some auxiliary variables that model the market impact cost. Indeed it holds

$$z_j \geq |x_j - x_j^0|$$

and

$$t_j \geq z_j^{3/2}.$$

Tag proceeding it should be mentioned that transaction costs of the form

$$c_j \geq z_j^{p/q}$$

where p and q are both integers and $p \geq q$ can be modelled using quadratic cones. See *[MOSEKApS12]* for details.

Example code

Listing 7.3 demonstrates how to compute an optimal portfolio when market impact cost are included using *Fusion*.

Listing 7.3: Implementation of model (7.10).

```python
def MarkowitzWithMarketImpact(n,mu,GT,x0,w,gamma,m):
    """
        Description:
            Extends the basic Markowitz model with a market cost term.

        Input:
            n: Number of assets
            mu: An n dimensional vector of expected returns
            GT: A matrix with n columns so (GT')*GT  = covariance matrix
            x0: Initial holdings
            w: Initial cash holding
            gamma: Maximum risk (=std. dev) accepted
            m: It is assumed that  market impact cost for the j'th asset is
               m_j|x_j-x0_j|^3/2

        Output:
           Optimal expected return and the optimal portfolio

    """

    with  Model("Markowitz portfolio with market impact") as M:

        #M.setLogHandler(sys.stdout)
```

```
        # Defines the variables. No shortselling is allowed.
        x = M.variable("x", n, Domain.greaterThan(0.0))

        # Additional "helper" variables
        t = M.variable("t", n, Domain.unbounded())
        z = M.variable("z", n, Domain.unbounded())
        v = M.variable("v", n, Domain.unbounded())

        #  Maximize expected return
        M.objective('obj', ObjectiveSense.Maximize, Expr.dot(mu,x))

        # Invested amount + slippage cost = initial wealth
        M.constraint('budget', Expr.add(Expr.sum(x),Expr.dot(m,t)), Domain.equalsTo(w+sum(x0)))

        # Imposes a bound on the risk
        M.constraint('risk', Expr.vstack(gamma,Expr.mul(GT,x)), Domain.inQCone())

        # z >= |x-x0|
        M.constraint('buy', Expr.sub(z,Expr.sub(x,x0)),Domain.greaterThan(0.0))
        M.constraint('sell', Expr.sub(z,Expr.sub(x0,x)),Domain.greaterThan(0.0))

        # t >= z^1.5, z >= 0.0. Needs two rotated quadratic cones to model this term
        M.constraint('ta', Expr.hstack(v,t,z),Domain.inRotatedQCone())
        M.constraint('tb', Expr.hstack(z,Expr.constTerm(n,1.0/8.0),v),\
                         Domain.inRotatedQCone())

        M.solve()

        print("\n-------------------------------------------------------------------------------
↪-----");
        print('Markowitz portfolio optimization with market impact cost')
        print("-------------------------------------------------------------------------------
↪---\n");
        print('Expected return: %.4e Std. deviation: %.4e Market impact cost: %.4e' % \
              (dot(mu,x.level()),gamma,dot(m,t.level())))

        return (dot(mu,x.level()), x.level())
```

The major new feature compared to the previous examples are

```
M.constraint('ta', Expr.hstack(v,t,z),Domain.inRotatedQCone())
```

and

```
M.constraint('tb', Expr.hstack(z,Expr.constTerm(n,1.0/8.0),v),\
                 Domain.inRotatedQCone())
```

In the first line the variables v, t and z are stacked horizontally which corresponds to creating a list of linear expressions where the j'th element has the form

$$\begin{bmatrix} v_j \\ t_j \\ z_j \end{bmatrix}$$

and finally each linear expression are constrained to be in rotated quadratic cone i.e.

$$2v_jt_j \geq z_j^2 \text{ and } v_j, t_j \geq 0.$$

Similarly the second line is equivalent to the constraint

$$\begin{bmatrix} z_j \\ 1/8 \\ v_j \end{bmatrix} \in Q_r^3$$

or equivalently

$$2z_j\frac{1}{8} \geq v_j^2 \text{ and } z_j \geq 0.$$

7.1.6 Transaction Costs

Now assume there is a cost associated with trading asset j and the cost is given by

$$T_j(\Delta x_j) = \begin{cases} 0, & \Delta x_j = 0, \\ f_j + g_j|\Delta x_j|, & \text{otherwise.} \end{cases}$$

Delta x_j is the change in the holding of asset j i.e.

$$\Delta x_j = x_j - x_j^0.$$

Hence, whenever asset j is traded a fixed cost of f_j has to be paid and a variable cost of g_j per unit traded. Given the assumptions about transaction costs in this section then problem (7.5) may be formulated as

$$\begin{array}{lllll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + \sum_{j=1}^n (f_j y_j + g_j z_j) & = & w + e^T x^0, & \\
 & [\gamma; G^T x] & \in & Q^{n+1}, & \\
 & z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
 & z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
 & z_j & \leq & U_j y_j, & j = 1, \ldots, n, \\
 & y_j & \in & \{0, 1\}, & j = 1, \ldots, n, \\
 & x & \geq & 0. &
\end{array} \tag{7.11}$$

First observe that

$$z_j \geq |x_j - x_j^0|$$

and hence z_j is bounded below by $|\Delta x_j|$. U_j is some a prior chosen upper bound on the amount of trading in asset j and therefore if $z_j > 0$ then $y_j = 1$ has to be the case. This implies that the transaction costs for the asset j is given by autonomous

$$f_j y_j + g_j z_j.$$

Example code

The following example code demonstrates how to compute an optimal portfolio when transaction costs are included.

Listing 7.4: Code solve problem (7.11).

```
def MarkowitzWithTransactionsCost(n,mu,GT,x0,w,gamma,f,g):
    """
        Description:
            Extends the basic Markowitz model with a market cost term.

        Input:
            n: Number of assets
            mu: An n dimensional vector of expected returns
            GT: A matrix with n columns so (GT')*GT  = covariance matrix
            x0: Initial holdings
            w: Initial cash holding
```

```
            gamma: Maximum risk (=std. dev) accepted
            f: If asset j is traded then a fixed cost f_j must be paid
            g: If asset j is traded then a cost g_j must be paid for each unit traded

        Output:
           Optimal expected return and the optimal portfolio

    """

    # Upper bound on the traded amount
    w0 = w+sum(x0)
    u = n*[w0]

    with Model("Markowitz portfolio with transaction costs") as M:
        #M.setLogHandler(sys.stdout)

        # Defines the variables. No shortselling is allowed.
        x = M.variable("x", n, Domain.greaterThan(0.0))

        # Additional "helper" variables
        z = M.variable("z", n, Domain.unbounded())
        # Binary variables
        y = M.variable("y", n, Domain.binary())

        #  Maximize expected return
        M.objective('obj', ObjectiveSense.Maximize, Expr.dot(mu,x))

        # Invest amount + transactions costs = initial wealth
        M.constraint('budget', Expr.add([ Expr.sum(x), Expr.dot(f,y),Expr.dot(g,z)] ), Domain.
↪equalsTo(w0))

        # Imposes a bound on the risk
        M.constraint('risk', Expr.vstack( gamma,Expr.mul(GT,x)), Domain.inQCone())

        # z >= |x-x0|
        M.constraint('buy', Expr.sub(z,Expr.sub(x,x0)),Domain.greaterThan(0.0))
        M.constraint('sell', Expr.sub(z,Expr.sub(x0,x)),Domain.greaterThan(0.0))
        # Alternatively, formulate the two constraints as
        #M.constraint('trade', Expr.hstack(z,Expr.sub(x,x0)), Domain.inQcone())

        # Constraints for turning y off and on. z-diag(u)*y<=0 i.e. z_j <= u_j*y_j
        M.constraint('y_on_off', Expr.sub(z,Expr.mulElm(u,y)), Domain.lessThan(0.0))

        # Integer optimization problems can be very hard to solve so limiting the
        # maximum amount of time is a valuable safe guard
        M.setSolverParam('mioMaxTime', 180.0)
        M.solve()

        print("\n-----------------------------------------------------------------------------
↪-----");
        print('Markowitz portfolio optimization with transactions cost')
        print("------------------------------------------------------------------------------
↪---\n");
        print('Expected return: %.4e Std. deviation: %.4e Transactions cost: %.4e' % \
              (dot(mu,x.level()),gamma,dot(f,y.level())+dot(g,z.level())))

        return (dot(mu,x.level()), x.level())
```

7.2 Primal Support-Vector Machine (SVM)

Machine-Learning (ML) has become a common widespread tool in many applications that affect our everyday life. In many cases, at the very core of these techniques there is an optimization problem. This case studies focuses on the Support-Vector Machine (SVM).

In words, the basic SVM model can be stated as:

> We are given a set of points m in a n-dimensional space, partitioned in two groups. Find, if any, the separating hyperplane of the two subsets with the largest margin, i.e. as far as possible from the points.

Mathematical Model

We must determine an hypeplane $w^T x = b$ that separate two sets of points leaving the largest margin possible. It can be proved that the margin is given by $2\|w\|$(see *[CV95]*).

Therefore, we need to solve the problem of maximizing $2\|w\|$ with respect of w, b with the constraints that points of the same class must lie on the same side of the hyperplane. Denoting with $x_i \in \mathbb{R}^n$ the i-th observation and assuming that each point is given a label $y_i \in \{-1, +1\}$, it is easy to see that the separation is equivalent to:

$$y_i(w^T x_i - b) \geq 1.$$

The separating hyperplane is the solution of the following optimization problem:

$$\begin{array}{ll} \text{minimize}_{b,w} & \frac{1}{2}\|w\|^2 \\ & y_i(w^T x_i - b) \geq 1 \quad i = 1, \ldots, m \end{array}$$

If a solution exists, w, b define the separating hyperplane and the sign of $w^T x - b$ can be used to decide the class in which a point x falls.

To allow more flexibility the soft-margin SVM classifier is often used instead. It allows for violation of the classification. To this extent a non-negative slack variable is added to each linear constraint and penalized in the objective function.

$$\begin{array}{lll} \text{minimize}_{b,w} & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^m \xi_i & \\ & y_i(w^T x_i - b) \geq 1 - \xi_i & i = 1, \ldots, m \\ & \xi_i \geq 0 & i = 1, \ldots, m \end{array}$$

In matrix form we have

$$\begin{array}{ll} \text{minimize}_{b,w,\xi} & \frac{1}{2}\|w\|^2 + C\mathbf{e}^T\xi \\ & y \star (Xw - b\mathbf{e}) + \xi \geq \mathbf{e} \\ & \xi \geq 0 \end{array}$$

where $\star$ denotes the component-wise product, and $\mathbf{e}$ a vector with all components equal to one. The constant $C \geq 0$ acts both as scaling factor and as weight. Varying C yields different trade-off between accuracy and robustness.

Implementing the matrix formulation of the soft-margin SVM in *Fusion* is very easy. We only need to cast the problem in conic form, which in this case only involves converting the quadratic term of the objective function in a conic constraint:

$$\begin{array}{ll} \text{minimize}_{b,w,\xi,t} & t + C\mathbf{e}^T\xi \\ & \xi + y \star (Xw - b\mathbf{e}) \geq \mathbf{e} \\ & (1, t, w) \in \mathcal{Q}_r^{n+2} \\ & \xi \geq 0 \end{array} \tag{7.12}$$

where $\mathcal{Q}_\nabla$ denotes a rotated cone of dimension $n + 2$.

Fusion implementation

We show now how implement model (7.12). Let assume now we are given an array `y` of labels, a matrix `X` where input data are stored row-wise and a set of values `CC` for we want to test. The implementation in *Fusion* of our conic model starts declaring the model class:

```
with Model() as M:
```

Note how we use the **using** statements to enforce the class scope (see Section *3.1.2*).

Then we proceed defining the variables

```
w =  M.variable('w' , n, Domain.unbounded())
t =  M.variable('t' , 1, Domain.unbounded())
b =  M.variable('b' , 1, Domain.unbounded())
xi = M.variable('xi', m, Domain.greaterThan(0.))
```

The conic constraint is obtained stacking the three values

```
M.constraint( Expr.vstack( 1., t, w) , Domain.inRotatedQCone())
```

Note how the dimension of the cone is deduced from the arguments. The relaxed classification constraints can be expressed using the built-in expressions available in *Fusion*. In particular, it is very helpful to

1. use the element wise multiplication to perform $y \star \cdot$, using the *Expr.mulElm* function;
2. construct a vector whose entries are all repetition of b by calling *Var.repeat*.

The results is

```
M.constraint(
    Expr.add(
        Expr.mulElm( y,
                     Expr.sub( Expr.mul(X,w), Var.repeat(b,m) )
                 ),
        xi
    ),
    Domain.greaterThan( 1. ) )
```

Finally, the objective function is defined as

```
M.objective(ObjectiveSense.Minimize, Expr.add( t, Expr.mul(C, Expr.sum(xi) ) ) )
```

Since our aim is to solve sequence of problem varying `C`, then we can simply iterates along those values changing the objective function:

```
for C in CC:
    M.objective(ObjectiveSense.Minimize, Expr.add( t, Expr.mul(C, Expr.sum(xi) ) ) )
    M.solve()
```

The overall code follows:

Listing 7.5: The code implementing model (7.12)

```
def primal_svm(m,n,X,y,CC):

    print("Number of data    : %d"%m)
    print("Number of features: %d"%n)

    with Model() as M:
```

```
        w =  M.variable('w' , n, Domain.unbounded())
        t =  M.variable('t' , 1, Domain.unbounded())
        b =  M.variable('b' , 1, Domain.unbounded())
        xi = M.variable('xi', m, Domain.greaterThan(0.))

        M.constraint(
            Expr.add(
                Expr.mulElm( y,
                             Expr.sub( Expr.mul(X,w), Var.repeat(b,m) )
                           ),
                xi
            ),
            Domain.greaterThan( 1. ) )


        M.constraint( Expr.vstack( 1., t, w) , Domain.inRotatedQCone())

        M.acceptedSolutionStatus(AccSolutionStatus.NearOptimal)

        print ('   c    | b       | w')

        for C in CC:
            M.objective(ObjectiveSense.Minimize, Expr.add( t, Expr.mul(C, Expr.sum(xi) ) ) )
            M.solve()

            try:
                cb = '{0:6} | {1:8f} | '.format(C,b.level()[0])
                wstar =' '.join([ '{0:8f}'.format(wi) for wi in w.level()])
                print (cb+wstar)
            except:
                pass;
```

Computational Tests

We show now few simple tests.

We generate random data composed by two sets of points using the following code:

```
CC=[ 500.0*i for i in range(10)]

m   = 50
n   = 3
seed= 0

random.seed(seed)


nump= random.randint(0,50)
numm= m - nump

y = [  1. for i in range(nump)] + \
    [ -1. for i in range(numm)]

mean = 1.
var = 1.

X=   [ [ random.gauss( mean,var) for f in range(n)  ]  for i in range(nump)] + \
     [ [ random.gauss(-mean,var) for f in range(n)  ]  for i in range(numm)]
```

As first tests, we generate two sets of random two dimensional points each from a Gaussian distribution: we use a set centered at $(1.0, 1.0)$ and another at $(-1.0, -1.0)$.

With a standard deviation $\sigma = 1/2$ we obtain a separable sets of points and for C we obtain the result in Fig. 7.1.

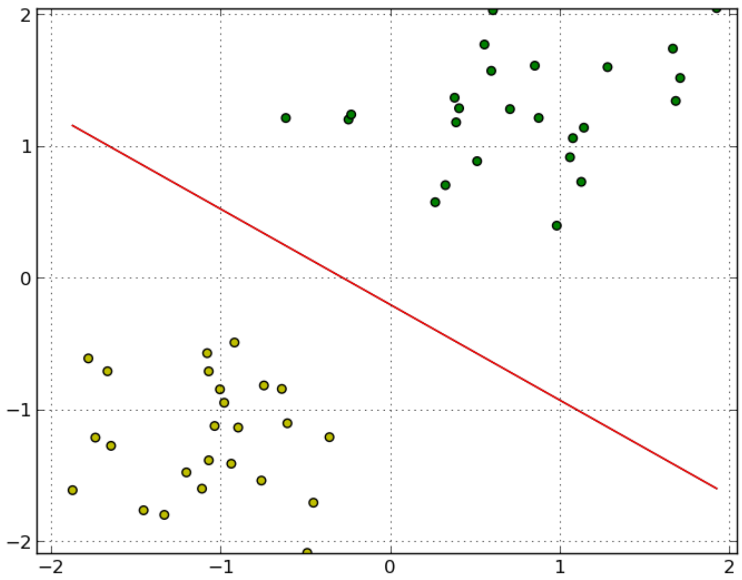


Fig. 7.1: Separating hyper plane for two group of points in two dimensions.

For $\sigma = 1$ separability is lost and we obtain the hyper plane as for Fig. 7.2.

7.3 2D Total Variation

This case studies is mainly based on the paper by Goldfarb and Yin *[GY05]*.

Mathematical Formulation

We are given a $n \times m$ grid and for each cell (i, j) an observed value f_{ij} that can expressed as

$$f_{ij} = u_{ij} + v_{ij},$$

where u_{ij} is the actual value and v_{ij} is the noise. The aim is to reconstruct f subtracting the noise from the observations.

We assume the 2-norm of the overall noise to be bounded: the corresponding constraint is

$$\|u - f\|_2 \leq \sigma$$

which translate in a simple conic quadratic constraint as

$$(\sigma, u - f) \in \mathcal{Q}$$

Then our aim is to minimized the change in both axis moving from one cell to the adjacent ones. To this end we define the adjacent differences vector as

$$\partial_{ij}^{+} = \left(\begin{array}{c} \partial_{ij}^{x} \\ \partial_{ij}^{y} \end{array} \right) = \left(\begin{array}{c} u_{i+1,j} - u_{i,j} \\ u_{i,j+1} - u_{i,j} \end{array} \right), \tag{7.13}$$

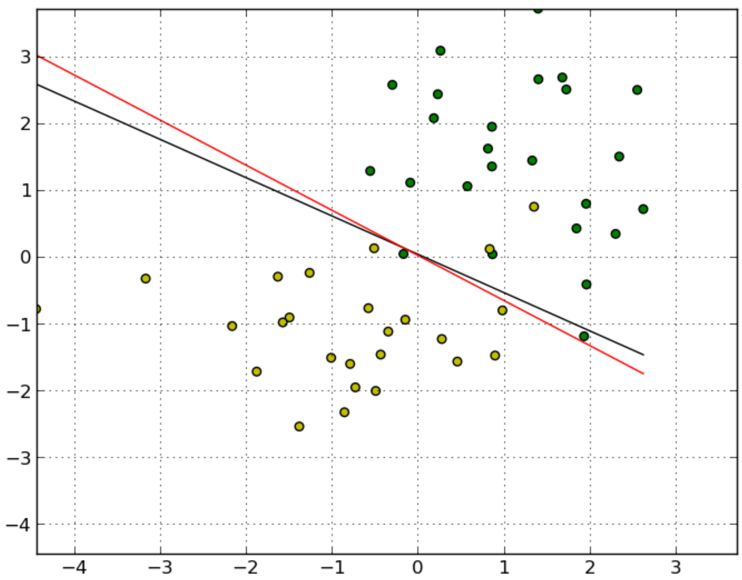
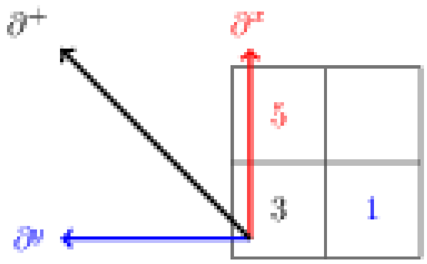


Fig. 7.2: Soft separating hyper plane for two group of points in two dimensions.



for each pair of cells $1 \leq i, j \leq n$. The idea is depicted in the following figure.

For each cells we want to minimize the norm of ∂^+_{ij}, and therefore we introduce auxiliary variables t_{ij} such that

$$t_{ij} \geq \|\partial^+_{ij}\|_2 \quad \forall i, j,$$

that we reformulate as

$$\left(t_{ij}, \partial^+_{ij}\right) \in \mathcal{Q} \quad \forall i, j,$$

and minimize the sum of all t_{ij}.

The complete model takes the form:

$$\begin{array}{lll} \min & \sum_{1\leq i,j\leq n} t_{ij} & \\ & \partial^+_{ij} = \left(u_{i+1,j} - u_{i,j}, u_{i,j+1} - u_{i,j}\right)^T & \forall 1 \leq i, j \leq n \\ & \left(t_{ij}, \partial^+_{ij}\right) \in \mathcal{Q} & \forall 1 \leq i, j \leq n \\ & \left(\sigma, \mathrm{vect}(u - f)\right) \in \mathcal{Q} & \\ & u_{i,j} \in [0, 1] & \forall 1 \leq i, j \leq n \end{array} \tag{7.14}$$

Implementation

The *Fusion* implementation of model (7.14) relies on the possibility of define variable and expression slices.

First of all we start creating the optimization model and variables `t` and `u`:

```
with Model('TV') as M:

    u= M.variable( [n+1,m+1], Domain.inRange(0.,1.0))
    t= M.variable( [n,m], Domain.unbounded())
```

Note how `u` is larger than the actual grid dimension to account for additional cells. Then we define a slice of `u` that contains the actual cells of the grid:

```
ucore=  u.slice([0,0],[n,m])
```

The next steps is to define the partial variation on each axis, as in (7.13):

```
deltax= Expr.sub( u.slice( [1,0] ,[n+1,m] ), ucore)
deltay= Expr.sub( u.slice( [0,1] ,[n,m+1] ), ucore)
```

Slices are in this case created on the fly as they are not going to be reused thereafter. Now we can set the conic constraints on the norm of the total variations. To this extent:

1. we proceed flattening `deltax,deltay` and `t` so that they become three one-dimensional arrays;
2. they can then be stacked horizontally using *Expr.hstack*, obtaining a matrix of expressions
3. each row of that matrix can be assigned to a rotated quadratic cone simply using *Domain.inRotatedQCone*.

All these steps can be condensed in the following line:

```
M.constraint( Expr.stack(2, t, deltax, deltay ), Domain.inQCone().axis(2) )
```

We need now to bound the norm of the error cell-wise. A conic constraint suffices using `f` as an one-dimensional array:

```
fmat = Matrix.dense(n,m,f)
M.constraint(Expr.vstack(sigma, Expr.flatten( Expr.sub( fmat, ucore ) )  ),
             Domain.inQCone() )
```

We only need to set the objective function as the sum of all t_{ij}'s:

```
M.objective(ObjectiveSense.Minimize, Expr.sum(t) )
```

The overall code follows.

Listing 7.6: The *Fusion* implementation of model (7.14).

```
##
# Copyright:  Copyright (c) MOSEK ApS, Denmark. All rights reserved.

#
# File:       total_variation.py
#
# Purpose:    Demonstrates how to solve a total
#             variation problem using the
#             Fusion API.
#
##
import sys
import mosek
from mosek.fusion import *
import numpy as np

def total_var(n,m,f,sigma):
    """
    2-dimensional model using sliding windows
    """
    with Model('TV') as M:

        u= M.variable( [n+1,m+1], Domain.inRange(0.,1.0))
        t= M.variable( [n,m], Domain.unbounded())

        ucore=  u.slice([0,0],[n,m])

        deltax= Expr.sub( u.slice( [1,0] ,[n+1,m] ), ucore)
        deltay= Expr.sub( u.slice( [0,1] ,[n,m+1] ), ucore)

        M.constraint( Expr.stack(2, t, deltax, deltay ), Domain.inQCone().axis(2) )

        fmat = Matrix.dense(n,m,f)
        M.constraint(Expr.vstack(sigma, Expr.flatten( Expr.sub( fmat, ucore ) )  ),
                     Domain.inQCone() )

        #boundary conditions are not actually needed
        #print('boundary conditions u_n+1,* = u_n,*  ...')
        #M.constraint( Expr.sub( u.slice( [n-1,0] ,[n,m] ), u.slice([n,0],[n+1,m]) ),
        #Domain.equalsTo(0.))
        #print('boundary conditions u_*,n+1 = u_*,n  ...')
        #M.constraint( Expr.sub( u.slice( [0,n-1] ,[n,m] ), u.slice([0,n],[n,m+1]) ),
        #Domain.equalsTo(0.))

        M.objective(ObjectiveSense.Minimize, Expr.sum(t) )
        M.setLogHandler(sys.stdout)

        M.solve()

        try:
            return u.slice([0,0],[n,m]).level()
```

```
        except:
            pass

    return None



if __name__ == '__main__':

    nrows = 50
    ncols = 50
    np.random.seed(0)
    ncells=  nrows*ncols
    sigma = 1.0

    #generate a random distribution among the grid
    u = np.random.uniform(0., 1., ncells)
    #add a gaussian noise
    f = u + np.random.normal(0., sigma, ncells)
    #trim spikes
    f= np.minimum( np.maximum(f, np.zeros( f.shape ) ), np.ones(f.shape) )

    u = total_var(nrows, ncols, f , sigma)

    deltas= [ abs(u[i]-f[i]) for i in range(ncells)]
    print( 'max deltas= ',max(deltas))
    print( 'min deltas= ',min(deltas))
```

7.4 Inner and outer Löwner-John Ellipsoids

In this section we show how to compute the Löwner-John *inner* and *outer* ellipsoidal approximations of a polytope.

This is possible because for a given polytope, the volume of the inscribed (or enclosing) ellipsoid is proportional to a the n-th power of the determinant of the a specific positive semidefinite matrix. Details are given in Section *7.4.3*.

7.4.1 Inner Löwner-John Ellipsoids

The inner ellipsoidal approximation to a polytope

$$S = \{x \in \mathbb{R}^n \mid Ax \leq b\}$$

maximizes the volume of the inscribed ellipsoid,

$$\{x \mid x = Cu + d, \|u\|_2 \leq 1\}.$$

The volume is proportional to $\det(C)^{1/n}$, so the problem can be solved as

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & t \leq \det(C)^{1/n} \\ & \|Ca_i\|_2 \leq b_i - a_i^T d, \ i = 1, \ldots, m \\ & C \succeq 0 \end{array} \tag{7.15}$$

which is equivalent to a mixed conic quadratic and semidefinite programming problem.

The source code follows in Listing 7.7. See also Section *7.4.3*.

Listing 7.7: *Fusion* implementation of model (7.15).

```
def lownerjohn_inner(A, b):
    '''
    The inner ellipsoidal approximation to a polytope

       S = { x \in R^n | Ax < b }.

    maximizes the volume of the inscribed ellipsoid,

       { x | x = C*u + d, || u ||_2 <= 1 }.

    The volume is proportional to det(C)^(1/n), so the
    problem can be solved as

      maximize         t
      subject to       t        <= det(C)^(1/n)
                  || C*ai ||_2 <= bi - ai^T * d,  i=1,...,m
                  C is PSD

    which is equivalent to a mixed conic quadratic and semidefinite
    programming problem.

    References:
    [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
   '''

    with Model("lownerjohn_inner") as M:
        M.setLogHandler(sys.stdout)
        m, n = len(A), len(A[0])

        # Setup variables
        t = M.variable("t", 1, Domain.greaterThan(0.0))
        C = M.variable("C", [n,n], Domain.unbounded())
        d = M.variable("d", n, Domain.unbounded())

        # (bi - ai^T*d, C*ai) \in Q, i=1..m
        M.constraint("qc", Expr.hstack(Expr.sub(b, Expr.mul(A,d)), Expr.mul(A,C.transpose())),
                     Domain.inQCone())
        # t <= det(C)^{1/n}
        det_rootn(M, C, t)

        # Objective: Maximize t
        M.objective(ObjectiveSense.Maximize, t)

        M.solve()

        C, d = C.level(), d.level()
        return ([C[i:i+n] for i in range(0,n*n,n)], d)
```

7.4.2 Outer Löwner-John Ellipsoids

The outer ellipsoidal approximation to a polytope given as the convex hull of a set of points

$$S = \mathrm{conv}\{x_1, x_2, \ldots, x_m\}$$

minimizes the volume of the enclosing ellipsoid,

$$\{x \mid \|P(x - c)\|_2 \leq 1\}$$

The volume is proportional to $\det(P)^{-1/n}$, so the problem can be solved as

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & t \geq \det(P)^{-1/n} \\ & \|Px_i + c\|_2 \leq 1, \; i = 1, \ldots, m \\ & P \succeq 0. \end{array} \tag{7.16}$$

The source code follows in Listing 7.8. See also Section *7.4.3*.

Listing 7.8: *Fusion* implementation of model (7.16).

```python
def lownerjohn_outer(x):
    '''
    The outer ellipsoidal approximation to a polytope given
    as the convex hull of a set of points

      S = conv{ x1, x2, ... , xm }

    minimizes the volume of the enclosing ellipsoid,

      { x | || P*(x-c) ||_2 <= 1 }

    The volume is proportional to det(P)^{-1/n}, so the problem can
    be solved as

      minimize         t
      subject to       t        >= det(P)^(-1/n)
                  || P*xi + c ||_2 <= 1,   i=1,...,m
                  P is PSD.

    References:
    [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
    '''

    with Model("lownerjohn_outer") as M:
        m, n = len(x), len(x[0])
        M.setLogHandler(sys.stdout)
        # Setup variables
        t = M.variable("t", 1, Domain.greaterThan(0.0))
        P = M.variable("P", [n,n], Domain.unbounded())
        c = M.variable("c", n, Domain.unbounded())

        # (1, P(*xi+c)) \in Q
        M.constraint("qc",
                     Expr.hstack(Expr.ones(m),
                                 Expr.sub(Expr.mul(x,P.transpose()),
                                          Var.reshape(Var.repeat(c,m), [m,2])
                                         )
                               ),
                     Domain.inQCone())

        # t <= det(P)^{1/n}
        det_rootn(M, P, t)

        # Objective: Maximize t
        M.objective(ObjectiveSense.Maximize, t)
        M.solve()

        P, c = P.level(), c.level()
        return ([P[i:i+n] for i in range(0,n*n,n)], c)
```

7.4.3 Bound on the Determinant Root.

Using SDP variables it is possible to bound the n-power of the determinant of a positive definite matrix, by modeling its hypograph:

$$C = \{(X, t) \in \mathcal{S}^n_+ \times \mathbb{R} | t <= det(X)^{1/n}\} \tag{7.17}$$

The set (7.17) can be modeled as the intersection of a semidefinite cone

$$[X, Z; Z^T Diag(Z)] \geq 0$$

and a number of rotated quadratic cones and affine hyperplanes,

$$t <= (Z11 * Z22 * ... * Znn)^{1/n},$$

which model the geometric mean hypograph

$$S = \{(x, t) \in \mathbb{R}^n \times \mathbb{R} | x \geq 0, t \leq \prod x_i (x1 \cdot x2 \cdot ... \cdot xn)^(1/n)\}$$

as the intersection of rotated quadratic cones and affine hyperplane.

See *[BenTalN01]* for further reading.

The *Fusion* implementation of the constraint on the n-th power of the root of the determinant is reported in Listing 7.9.

Listing 7.9: Bound on the root determinant n-th power, see (7.17).

```
def det_rootn(M, X, t):
    '''

     Purpose: Models the hypograph of the n-th power of the
     determinant of a positive definite matrix. See [1,2] for more details.

       The convex set (a hypograph)

       C = { (X, t) \in S^n_+ x R |  t <= det(X)^{1/n} },

       can be modeled as the intersection of a semidefinite cone

       [ X, Z; Z^T Diag(Z) ] >= 0

       and a number of rotated quadratic cones and affine hyperplanes,

       t <= (Z11*Z22*...*Znn)^{1/n}  (see geometric_mean).

       References:
       [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
       [2] "MOSEK modeling manual", 2013
    '''
    n = int(sqrt(X.size()))

    # Setup variables
    Y = M.variable(Domain.inPSDCone(2*n))

    # Setup Y = [X, Z; Z^T , diag(Z)]
    Y11 = Y.slice([0, 0], [n, n])
    Y21 = Y.slice([n, 0], [2*n, n])
    Y22 = Y.slice([n, n], [2*n, 2*n])

    M.constraint( Expr.sub(Expr.mulElm( Matrix.eye(n) ,Y21), Y22), Domain.equalsTo(0.0) )
    M.constraint( Expr.sub(X, Y11), Domain.equalsTo(0.0) )

    # t^n <= (Z11*Z22*...*Znn)
    geometric_mean(M, Y22.diag(), t)
```

The code is relies on a recursive implementation of the constraint on the geometric mean, as in Listing 7.10.

Listing 7.10: Bound on the geometric mean.

```
def geometric_mean(M,x,t):
  '''
  Models the convex set

    S = { (x, t) \in R^n x R | x >= 0, t <= (x1 * x2 * ... * xn)^(1/n) }

  as  the intersection of rotated quadratic cones and affine hyperplanes.
  see [1, p. 105] or [2, p. 21].  This set can be interpreted as the hypograph of the
  geometric mean of x.

  We illustrate the modeling procedure using the following example.
  Suppose we have

     t <= (x1 * x2 * x3)^(1/3)

  for some t >= 0, x >= 0. We rewrite it as

     t^4 <= x1 * x2 * x3 * x4,    x4 = t

  which is equivalent to (see [1])

     x11^2 <= 2*x1*x2,    x12^2 <= 2*x3*x4,

     x21^2 <= 2*x11*x12,

     sqrt(8)*x21 = t, x4 = t.

    References:
    [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
    [2] "MOSEK modeling manual", 2013

  '''


  def rec(x):
    n = x.getShape().dim(0)
    if n > 1:
      y = M.variable(int(n//2), Domain.unbounded())
      M.constraint(Var.hstack(Var.reshape(x, [n//2,2]), y), Domain.inRotatedQCone())
      return rec(y)
    else:
      return x

  n = x.getShape().dim(0)
  l = int(ceil(log(n, 2)))
  m = int(2**l) - n

  # if size of x is not a power of 2 we pad it:
  if m > 0:
    x_padding = M.variable(m,Domain.unbounded())

    # set the last m elements equal to t
    M.constraint(Expr.sub(x_padding, Var.repeat(t,m)), Domain.equalsTo(0.0))

    x = Var.vstack(x,x_padding)

  M.constraint(Expr.sub(Expr.mul(2.0**(l/2.0), t),rec(x)), Domain.equalsTo(0.0))
```

7.5 Nearest Correlation Matrix Problem

In the nearest correlation problem we are given a symmetric matrix A and we wish to find the closest correlation matrix, with respect to a given norm. A correlation matrix must be a positive-semidefinite matrix. In the next sections we will show two typical approaches:

- use the *Frobenius norm*,
- use the so-called *nuclear norm*.

In both cases we can exploit the symmetry of the correlation matrix to reduce the problem dimension. We will make use of the following mapping from a symmetric matrix to vector containing the (scaled) lower triangular part of the matrix,

$$\begin{array}{ll} \mathrm{vec}(M) : \mathbb{R}^{n\times n} \rightarrow \mathbb{R}^{n(n+1)/2} & \\ & \\ \mathrm{vec}(M)_k = M_{ij} & \text{for } k = i(i+1)/2 + j, i = j \\ \mathrm{vec}(M)_k = \sqrt{2}M_{ij} & \text{for } k = i(i+1)/2, i < j. \end{array} \tag{7.18}$$

In Listing 7.11 the *Fusion* implementation of *vec*.

Listing 7.11: Implementation of function *vec* in (7.18).

```python
def vec(e):
    """
    Assuming that e is an NxN expression, return the lower triangular part as a vector.
    """
    N = e.getShape().dim(0)

    rows = [i for i in range(N) for j in range(i,N)]
    cols = [j for i in range(N) for j in range(i,N)]
    vals = [ 2.0**0.5  if i!=j else 1.0 for i in range(N)  for j in range(i,N)]

    return Expr.flatten(Expr.mulElm(e, Matrix.sparse(N,N,rows,cols,vals)))
```

7.5.1 Nearest Correlation with Frobeniues Norm

In this section we use the Frobineus norm, i.e. the nearest correlation matrix is given by

$$X^\star = \arg\min_{x\in S} \|A - X\|_F$$

where

$$S = \{X \in \mathbb{R}^{n\times n} \mid X \succeq 0, \mathbf{diag}(X) = e\}.$$

To exploit symmetry of $A - X$ we use the *vec* mapping in (7.18). We then get an optimization problem with both conic quadratic and semidefinite constraints,

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & (t, \mathrm{vec}(A - X)) \in \mathcal{Q} \\ & \mathbf{diag}(X) = e \\ & X \succeq 0, \end{array} \tag{7.19}$$

Source code: Nearest correlation

The code implementing problem (7.19) is reported in Listing 7.12.

Listing 7.12: Implementation of problem (7.19).

```
def nearestcorr(A):

    N = A.numRows()

    # Create a model with the name 'NearestCorrelation

    with Model("NearestCorrelation") as M:

      # Setting up the variables
      X = M.variable("X", Domain.inPSDCone(N))
      t = M.variable("t", 1, Domain.unbounded())


      # (t, vec (A-X)) \in Q
      v = vec( Expr.sub(A,X) )
      M.constraint("C1", Expr.vstack(t, v ), Domain.inQCone() )

      # diag(X) = e
      M.constraint("C2",X.diag(), Domain.equalsTo(1.0))
#      M.setLogHandler(sys.stdout)
      # Objective: Minimize t
      M.objective(ObjectiveSense.Minimize, t)

      # Solve the problem
      M.solve()

      return X.level(),t.level()
```

We use the following input

Listing 7.13: Input for the nearest correlation problem.

```
N = 5
A = Matrix.dense(N,N,[ 0.0,  0.5,  -0.1,  -0.2,   0.5,
                      0.5,  1.25, -0.05, -0.1,   0.25,
                     -0.1, -0.05,  0.51,  0.02, -0.05,
                     -0.2, -0.1,   0.02,  0.54, -0.1,
                      0.5,  0.25, -0.05, -0.1,   1.25])
```

The expected out is the following (small differences may apply):

```
X =
[[ 1.          0.50001941 -0.09999994 -0.20000084  0.50001941]
 [ 0.50001941  1.         -0.04999551 -0.09999154  0.24999101]
 [-0.09999994 -0.04999551  1.          0.01999746 -0.04999551]
 [-0.20000084 -0.09999154  0.01999746  1.         -0.09999154]
 [ 0.50001941  0.24999101 -0.04999551 -0.09999154  1.        ]]
```

7.5.2 Nearest Correlation with Nuclear-norm Penalty

This is a variation of the nearest correlation matrix, where we estimate a correlation matrix $X \geq 0$, where $X - \mathbf{diag}(w) \geq 0$ has low rank induced by a nuclear norm constraint, and $w \geq 0$.

We solve the problem

$$
\begin{array}{ll}
\text{minimize} & \left\|X + \mathbf{diag}(w) - A\right\|_F + \gamma \mathrm{Tr}(X) \\
\text{subject to} & \\
 & X \succeq 0, w \geq 0.
\end{array}
$$

Again we can exploit symmetry of $A - X$ using the *vec* mapping in (7.18). We then get an optimization problem with both conic quadratic and semidefinite constraints.

$$\begin{array}{lll} \text{minimize} & t + \gamma \mathrm{Tr}(X) & \\ \text{subject to} & (t, \mathrm{vec}(X + \mathbf{diag}(w) - A)) & \in \mathcal{Q} \\ & X \in \mathcal{S}_+, w \geq 0 & \end{array}$$

The source code for this example follows in Listing 7.14.

Listing 7.14: Nearest correlation with nuclear norm.

```
def nearestcorr_nucnorm(A,gammas):
    N = A.numRows()
    with Model("NucNorm") as M:
      # Setup variables
      t = M.variable("t", 1, Domain.unbounded())
      X = M.variable("X", N, Domain.inPSDCone())
      w = M.variable("w", N, Domain.greaterThan(0.0))

      # D = diag(w)
      D =   Expr.mulElm( Matrix.eye(N), Var.repeat(w,1,N) )
      # (t, vec (X + D - A)) in Q
      M.constraint( Expr.vstack( t, vec(Expr.sub(Expr.add(X, D), A)) ),
                    Domain.inQCone() )

      #M.setLogHandler(sys.stdout)

      d = [0.0]*int(N)
      result = []

      for g in gammas:

          # Objective: Minimize t + gamma*Tr(X)
          M.objective(ObjectiveSense.Minimize, Expr.add(t, Expr.mul(g, Expr.sum(X.diag()) ) ))
          M.solve()

          Xl = X.level()
          wl = w.level()
          r = ([ 2*(A.get(j,i) - Xl[i+j*N])*(A.get(j,i) - Xl[i+j*N])
                for j in range(N) for i in range(j+1,N) ] +
               [(A.get(i,i) - Xl[i+i*N] - wl[i])*(A.get(i,i) - Xl[i+i*N] - wl[i])
                for i in range(N)])

          LinAlg.syeig(mosek.uplo.lo,N, X.level(), d)

          result.append((g, sqrt(sum(r)), sum([d[i]>1e-6 for i in range(N)]), X.level() ))

      return result
```

We feed **MOSEK** with the same input as in Section *7.5.1*. The problem is solved for a range of *gamma* values, to show how the penalty term helps achieve low rank solution. To this extent we report both the rank of the solution and the norm residual. The rank is computed using the *LinAlg.syeig* available in the **MOSEK**.

```
--- Nearest Correlation with Nuclear Norm---
gamma=0.000000, res=3.076163e-01, rank=4
gamma=0.100000, res=4.251692e-01, rank=2
gamma=0.200000, res=5.112082e-01, rank=1
gamma=0.300000, res=5.298432e-01, rank=1
gamma=0.400000, res=5.592686e-01, rank=1
gamma=0.500000, res=6.045702e-01, rank=1
gamma=0.600000, res=6.764402e-01, rank=1
```

```
gamma=0.700000, res=8.009913e-01, rank=1
gamma=0.800000, res=1.062385e+00, rank=1
gamma=0.900000, res=1.129513e+00, rank=0
gamma=1.000000, res=1.129513e+00, rank=0
```

7.6 Semidefinite Relaxation of MIQCQP Problems

In this case studies we will discuss a fairly common application for Semidefinite Optimization: to define continuous coninc relaxation of Mixed-Integer optimization problems.

We will focus on problems of the form:

$$\begin{array}{ll} \min & x^T P x + 2q^T x \\ & x \in \mathbb{Z}^n \end{array} \tag{7.20}$$

where $q \in \mathbb{R}^n$ and $P \in \mathcal{S}_+^{n \times n}$. There are many important problems that can be reformulated as problem (7.20):

- *integer least squares problem* $\min \|Ax - b\|_2^2, \text{s.t.} x \in \mathbb{Z}^n$
- *closest vector problem* $\min \|v - z\|_2, \text{s.t.} z \in \{Bx | x \in \mathbb{Z}^n\}$

Following *[PB15]*, we can derive a continuous conic model. We first relax the integrality constraint

$$\begin{array}{ll} \min & x^T P x + 2q^T x \\ & x_i(x_i - 1) \geq 0 \quad i = 1, \ldots, n. \end{array}$$

The last constraint is still non-convex. We introduce a new variable $X \in \mathbb{R}^{n \times n}$, such that $X = x \cdot x^T$. The last constraint then reads

$$\mathbf{diag}(X) - x \geq 0,$$

and with few passages we can write

$$\begin{array}{ll} \min & \mathbf{Tr}(PX) + 2q^T x \\ & \mathbf{diag}(X) \geq x \\ & X = x \cdot x^T. \end{array}$$

To get a conic problem we relax the last constraint and apply the Schur complement. The final relaxation follows:

$$\begin{array}{ll} \min & \mathbf{Tr}(PX) + 2q^T x \\ & \mathbf{diag}(X) \geq x \\ & \left[\begin{array}{cc} X & x \\ x^T & 1 \end{array} \right] \in \mathcal{S}_+^n. \end{array} \tag{7.21}$$

We refer to *[PB15]* for more details.

Fusion Implementation

Implementing model (7.21) in *Fusion* is very simple. We assume we are given as input n, P and q. Then we proceed creating the optimization model

```
with Model() as M:
```

The important step is to define a single PSD variable

$$Z = \begin{bmatrix} X & x \\ x^T & 1 \end{bmatrix} \in \mathcal{S}_+^{n+1}.$$

Our code will create Z and two slices that corresponds to X, x:

```
Z = M.variable(n+1, Domain.inPSDCone())

X= Z.slice([0,0],[n,n])
x= Z.slice([0,n],[n,n+1])
```

The constraints are declared as follow

```
M.constraint( Expr.sub(X.diag(),x), Domain.greaterThan(0.) )

M.constraint( Z.index(n,n), Domain.equalsTo(1.))
```

The objective function uses several available linear expressions:

```
M.objective( ObjectiveSense.Minimize, Expr.add(
    Expr.sum( Expr.mulElm( Matrix.dense(n,n,P), X ) ),
    Expr.mul(2.0, Expr.dot( x, q) )
) )
```

Note that the *trace* operator is not directly available in *Fusion*, but its definition can be easily used instead.

The complete code follows in Listing 7.15.

Listing 7.15: *Fusion* implementation of model (7.21).

```
def miqcqp_sdo_relaxation(n,P,q):
    with Model() as M:

        Z = M.variable(n+1, Domain.inPSDCone())

        X= Z.slice([0,0],[n,n])
        x= Z.slice([0,n],[n,n+1])

        M.constraint( Expr.sub(X.diag(),x), Domain.greaterThan(0.) )

        M.constraint( Z.index(n,n), Domain.equalsTo(1.))

        M.objective( ObjectiveSense.Minimize, Expr.add(
            Expr.sum( Expr.mulElm( Matrix.dense(n,n,P), X ) ),
            Expr.mul(2.0, Expr.dot( x, q) )
        ) )

        M.solve()
```

Numerical Examples

We present now some simple numerical experiments. The input data are generate following again *[PB15]*.

1. We generate a matrix $A \in \mathbb{R}^{m \times n}$, such that whose entries are normally distributed, i.e. $A_{ij} = \mathcal{N}(0, 1)$
2. define $P = AA^T$

3. generate a vector x_s whose entries are random number uniformly distributed in $[0, 1]$.
4. define $q = -Px_s$

These linear algebra operations are conveniently performed using the BLAS and LAPACK routines available through the Optimizer API for Python.

```
# problem dimensions
n = 50
m = 2*n

# problem data
A =  [random.gauss(0., 1.0) for i in range(n*m)]
xs = [random.uniform(0., 1.0)  for i in range(n)]

P =[0.]*(n*n)
q =[0.]*n

mosek.LinAlg.syrk(
    mosek.uplo.lo,
    mosek.transpose.no,
    n,m,
    1.0, A, 0., P)

#must fill P upper triangular
for j in range(n):
    for i in range(j+1,n):
        P[j*n+i] = P[i*n+j]

#q = -P xs
mosek.LinAlg.gemv(
    mosek.transpose.no,
    n,
    n,
    -1.0,
    P,
    xs,
    0.,
    q)

miqcqp_sdo_relaxation(n,P,q)
```

7.7 SUDOKU

SUDOKU is a famous simple yet mind-blowing simple game. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called *boxes*, *blocks*, *regions*, or *sub-squares*) contains all of the digits from 1 to 9. For more information see http://en.wikipedia.org/wiki/Sudoku. Here is a simple example:

In a more general setting we are given a grid of dimension $n \times n$, with $n = m^2, m \in \mathbb{N}$. Each cell (i, j) must be filled with an integer $y_{ij} \in [1, n]$. Along each row and each column there must be no repetitions. No repetitions are allowed also in each sub-grid with corners $\{(mt + 1), (ml + 1), (m(t + 1)), (m(l + 1))\}$, for $t, l = 0, \ldots, m - 1$.

In general, each SUDOKU instance comes along with some values already determined. The purpose of that values is:

				4				
	5	8			3			
	1		2	8		9		
	7	3	1			8	4	
	4	1			9	2	7	
		4		6	5		8	
			4			1	6	
				9				

A simple unsolved Sudoku

3	8	2	5	4	7	6	9	2
4	5	8	9	1	3	7	2	6
7	1	5	2	8	6	9	3	4
6	7	3	1	5	2	8	4	9
9	2	6	8	7	4	5	1	3
8	4	1	6	3	9	2	7	5
1	9	4	7	6	5	3	8	2
5	3	9	4	2	8	1	6	7
2	6	7	3	9	1	4	5	8

The solution

- reduce the complexity of the game removing symmetries and guiding the initial moves of the player;
- ensure that there will be a unique solution.

The latter point requires a careful selection of the given cells, that is beyond the scope of this post. We only provide the model with the set F randomly generated. We represent such set in as list of triplets (i, j, v).

Note that SUDOKU is a **feasibility** problem. A typical Integer Programming formulation is straightforward: let x_{ijk} be a binary variable that takes 1 if k is put in cell (i, j). Then we look for a feasible solution of the system of constraints:

SUDOKU has been also a nice problem to fiddle with for many researchers in the optimization and related community. Indeed, its simple structure and the the easy way in which the results can be tested, makes it a perfect test problem.

SUDOKU is a typical assignment problem. Its constraints are commonly found in optimization problems about scheduling, resource allocations.

We will approach SUDOKU as a standard integer linear program, and we will show how easily and elegantly it can be implemented in *Fusion*.

Mathematical Formulation

In this section we formulate SUDOKU as a mixed-integer linear optimization problem. Let's introduce a binary variable x_{ijk} that takes 1 if the the digit k is put in the cell (i, j), or 0 otherwise. We must ask that for each cell only one digit is selected, i.e.

$$\sum_{k=0}^{n-1} x_{ijk} = 1 \qquad i, j = 0, \ldots, n-1 \tag{7.22}$$

Similar constraints can be used to force each digit to appears only once in each row or column

$$\begin{array}{ll} \sum_{i=0}^{n-1} x_{ijk} = 1 & j, k = 0, \ldots, n-1 \\ \sum_{j=0}^{n-1} x_{ijk} = 1 & i, k = 0, \ldots, n-1 \end{array} \tag{7.23}$$

To force a digit to appears only once in each sub-grid we can use the following

$$\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1 \qquad k = 0, \ldots, n-1 \text{ and } t, l = 0, \ldots, m-1 \tag{7.24}$$

For each given cell (i, j) in F we must set the corresponding value k, i.e.

$$x_{ijk} = 1.$$

Summarizing, and considering that there is no objective function to minimize, the optimization model for the SUDOKU problem takes the form

$$
\begin{array}{ll}
\min 0 & \\
\text{s.t.} & \\
\sum_{i=0}^{n-1} x_{ijk} = 1 & j,k = 0,\ldots,n-1 \\
\sum_{j=0}^{n-1} x_{ijk} = 1 & i,k = 0,\ldots,n-1 \\
\sum_{k=0}^{n-1} x_{ijk} = 1 & i,j = 0,\ldots,n-1 \\
\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1 & k = 0,\ldots,n-1 \text{ and} \\
 & t,l = 0,\ldots,m-1 \\
 & \\
x_{ijk} = 1 & \forall (i,j,k) \in F
\end{array}
\tag{7.25}
$$

Implementation with *Fusion*

The implementation in *Fusion* is straightforward. First, we represent the x variable using a three dimensional *Fusion* variable:

```
x= M.variable([n,n,n], Domain.binary())
```

Then we can define constraints (7.22) and (7.23) simply using the *Expr.sum* operator, that allows to sum the elements of an expression (in this case a the variable itself) along arbitrary dimensions. The code reads:

```
#each value only once per dimension
for d in range(m):
  M.constraint( Expr.sum(x,d), Domain.equalsTo(1.) )
```

The last set of constraints (7.24) , i.e. the sum over block, needs a little more effort: we must loop over all blocks and select the proper slice:

```
#each number must appear only once in a block
for k in range(n):
  for i in range(m):
     for j in range(m):
          M.constraint( Expr.sum( x.slice([i*m,j*m,k], [(i+1)*m,(j+1)*m, k+1]) ),
                        Domain.equalsTo(1.))
```

To set the triplets given in the set F we can use the *Variable.pick* method that returns a one dimensional view of an arbitrary set of elements of the variable.

```
M.constraint( x.pick(fixed), Domain.equalsTo(1.0) )
```

SUDOKU: the complete example code.

The complete code for the SUDOKU problem is shown in Listing 7.16.

Listing 7.16: *Fusion* implementation to solve SUDOKU.

```
##
# Copyright:  Copyright (c) MOSEK ApS, Denmark. All rights reserved.

#
#  File:      sudoku.py
#
#  Usage:  python sudoku.py
#
##
```

```python
import sys
import mosek
from mosek.fusion  import *

def print_solution(m,x):
    n=m*m
    print("\n")
    for i in range(n):
        ss= ""
        for j in range(n):
            if j%m==0: ss=ss+ " |"

            for k in range(n):
                if x.index([i,j,k]).level()>0.5:
                    ss=ss+" "+str(k+1)
                    break;

        print (ss+' |')
        if (i+1)%m==0:
            print ("\n")


def main():

   m= 3
   n= m*m

   hr_fixed= [ [1,5,4],\
               [2,2,5], [2,3,8], [2,6,3],\
               [3,2,1], [3,4,2], [3,5,8], [3,7,9],\
               [4,2,7], [4,3,3], [4,4,1], [4,7,8], [4,8,4],\
               [6,2,4], [6,3,1], [6,6,9], [6,7,2], [6,8,7],\
               [7,3,4], [7,5,6], [7,6,5], [7,8,8],\
               [8,4,4], [8,7,1], [8,8,6],\
               [9,5,9]
   ]

   fixed= [ [f[0]-1,f[1]-1,f[2]-1] for f in hr_fixed]

   with Model('SUDOKU') as M:
      x= M.variable([n,n,n], Domain.binary())

      #each value only once per dimension
      for d in range(m):
        M.constraint( Expr.sum(x,d), Domain.equalsTo(1.) )

      #each number must appear only once in a block
      for k in range(n):
        for i in range(m):
           for j in range(m):
                M.constraint( Expr.sum( x.slice([i*m,j*m,k], [(i+1)*m,(j+1)*m, k+1]) ),
                              Domain.equalsTo(1.))

      M.constraint( x.pick(fixed), Domain.equalsTo(1.0) )

      M.setLogHandler(sys.stdout)

      M.solve()

      #print the solution, if any...
      if M.getPrimalSolutionStatus() in [SolutionStatus.Optimal, SolutionStatus.NearOptimal]:
          print_solution(m,x)
```

```python
        else:
            print ("No solution found!")

if __name__ == '__main__':
    main()
```

The problem instance corresponding to Fig. 7.7 is hard-coded for sake of simplicity. It will produce the following output

```
Computer
  Platform               : Linux/64-X86
  Cores                  : 20

Problem
  Name                   : SUDOKU
  Objective sense        : min
  Type                   : LO (linear optimization problem)
  Constraints            : 296
  Cones                  : 0
  Scalar variables       : 729
  Matrix variables       : 0
  Integer variables      : 729

Optimizer started.
Mixed integer optimizer started.
Threads used: 20
Presolve started.
Presolve terminated. Time = 0.00
Presolved problem: 206 variables, 154 constraints, 683 non-zeros
Presolved problem: 0 general integer, 206 binary, 0 continuous
Clique table size: 154
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(%)  TIME
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
0        1        0        0        NA                   -0.0000000000e+00    NA          0.0
Cut generation started.
0        2        0        0        NA                   -0.0000000000e+00    NA          0.0
Cut generation terminated. Time = 0.00
0        3        1        0        0.0000000000e+00     -0.0000000000e+00    0.00e+00    0.0
An optimal solution satisfying the relative gap tolerance of 1.00e-02(%) has been located.
The relative gap is 0.00e+00(%).
An optimal solution satisfying the absolute gap tolerance of 0.00e+00 has been located.
The absolute gap is 0.00e+00.

Objective of best integer solution : 0.000000000000e+00
Best objective bound               : -0.000000000000e+00
Construct solution objective       : Not employed
Construct solution # roundings     : 0
User objective cut value           : 0
Number of cuts generated           : 18
  Number of Gomory cuts            : 18
Number of branches                 : 0
Number of relaxations solved       : 3
Number of interior point iterations: 4
Number of simplex iterations       : 39
Time spend presolving the root     : 0.00
Time spend in the heuristic        : 0.00
Time spend in the sub optimizers   : 0.00
  Time spend optimizing the root   : 0.01
```

```
Mixed integer optimizer terminated. Time: 0.04

Optimizer terminated. Time: 0.04

Integer solution solution summary
  Problem status  : PRIMAL_FEASIBLE
  Solution status : INTEGER_OPTIMAL
  Primal.  obj: 0.0000000000e+00    Viol.  con: 0e+00    var: 0e+00    itg: 0e+00


 | 3 8 2 | 5 4 7 | 6 9 1 |
 | 4 5 8 | 9 1 3 | 7 2 6 |
 | 7 1 5 | 2 8 6 | 9 3 4 |


 | 6 7 3 | 1 5 2 | 8 4 9 |
 | 9 2 6 | 8 7 4 | 5 1 3 |
 | 8 4 1 | 6 3 9 | 2 7 5 |


 | 1 9 4 | 7 6 5 | 3 8 2 |
 | 5 3 9 | 4 2 8 | 1 6 7 |
 | 2 6 7 | 3 9 1 | 4 5 8 |
```

7.8 Multi-processors Scheduling

In this case of study we consider a simple scheduling problem in which a set of jobs must be assigned to a set of identical machines. We want to minimize the makespan of the overall processing, i.e. the latest machine termination time.

The main aims of this case study are

- show how to define a Integer Linear Programming model,
- take advantage of *Fusion* operators to compactly express set of constraitns,
- provide to the solver an incumbent integer feasible solution.

Mathematical formulation

We are given a set of jobs J with $|J| = n$ to be assigned to a set M of identical machines with $|M| = m$. Each job $j \in J$ has a processing time $T_j > 0$ and can be assigned to any machine. Our aim is to find the job scheduling that minimizes the overall makespan, i.e. the maximum complation time among all machiens.

Formally, we introduce a binary variable x_{ij} that takes value one if the job j is assigned to the machine i, zero otherwise. The only constraint we need to set is the requirement that a job must be assigned to a single machine. The optimization model takes the following form:

$$
\begin{array}{ll}
\min \max_{i \in M} \sum_{j \in J} T_j x_{ij} & \\
\text{s.t.} & \\
\sum_{i \in M} x_{ij} = 1 & j \in J \\
x_{ij} \in \{0,1\} & \forall i \in M, j \in J
\end{array}
\tag{7.26}
$$

Model (7.26) can be easily transformed in an integer linear programing model as the following

$$
\begin{array}{ll}
\min t & \\
\text{s.t.} & \\
\sum_{i\in M} x_{ij} = 1 & j \in J \\
t \geq \sum_{j\in J} T_j x_{ij} & i \in M \\
 & \\
x_{ij} \in \{0,1\} & \forall i \in M, j \in J.
\end{array}
\tag{7.27}
$$

The implementation of this model in *Fusion* is straightforward:

```
with Model('Multi-processor scheduling') as M:

    x= M.variable('x', [m,n], Domain.binary())

    t= M.variable('t',1, Domain.unbounded())

    M.constraint( Expr.sum(x,0), Domain.equalsTo(1.) )

    M.constraint( Expr.sub( Var.repeat(t,m), Expr.mul(x,T) ) , Domain.greaterThan(0.) )

    M.objective( ObjectiveSense.Minimize, t )
```

Most of the code is self-explaining. The only critical point is

```
M.constraint( Expr.sub( Var.repeat(t,m), Expr.mul(x,T) ) , Domain.greaterThan(0.) )
```

that implements the set of constraints

$$t \geq \sum_{j\in J} T_j x_{ij} \quad i \in M$$

To fit in *Fusion* we restate the constraints as

$$t - \sum_{j\in J} T_j x_{ij} \geq 0 \quad i \in M$$

which corresponds in matrix form to

$$t\mathbf{1} - xT \geq 0. \tag{7.28}$$

The function *Var.repeat* creates a vector of lenght m, which is what is stated in (7.28). The same results can be obtained as matrix multiplication, i.e. using *Expr.mul*, but in this particular case *Var.repeat* is faster as it does only a logical operation.

Longest Processing Time first rule (LPT)

The multiprocessor scheduling is known to be an NP-complete problem (see *[GJ79]*). Neverthless there are effective heuristics, with proven worst case bound, that are able to provided a good integer solution quickly. In particular, we will use the so-called *Longest Processing Time first* rule (LPT, proposed in *[Gra69]*).

The informal algorithm sketch is the following:

- while M is not empty do
  - let `k` the machine with the smallest load so far
  - let `i` be the job in `M` with the longest completion time
  - assign job `i` to machine `k`
  - update machine `k` load

- remove `i` from `M`

This simple algorithm is a $1/3(4-1/m)$ approximation. So for $m = 1$ we get the optimal solution (indeed there is no choices with a single machine); for $m \to \infty$ the approximation tends to its worst case of $4/3$ (againg see *[Gra69]*).

A simple implementation is given below.

```
#LPT heuristic
schedule=[0. for i in range(m)]
init= [0. for i in range(n*m)]

for i in range(n):
    mm= schedule.index(min(schedule))
    schedule[mm]+= T[i]
    init[n*mm+i]=1.
```

An efficient implementation of the LPT rule is beyond the scope of this section. The important is that the scheduling produced by the LPT algorithm can be used as incumbent solution for the **MOSEK** mixed-integer linear programming solver. The availability of an integer feasible solution can significantly improve the performance of the solver.

To input the solution we only need to use the *Variable.setLevel* method, as shown below

```
x.setLevel(init)
```

The effect of the availability of an incumbent solution can be easily seen looking at the solver output.

For instance, let's consider the following input

Running **MOSEK** the solution is the following

```
initial solution:
('M', 0, [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0])
('M', 1, [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0])
('M', 2, [0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0])
('M', 3, [0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0])
MOSEK solution:
('M', 0, [0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0])
('M', 1, [1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
('M', 2, [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0])
('M', 3, [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0])
```

The running is very short. Without initial solution it much higher.

The complete code follows.

Listing 7.17: Complete code for LPT scheduling example.

```
##
#    Copyright:  Copyright (c) MOSEK ApS, Denmark. All rights reserved.

#
#    File:     lpt.py
#
#    Purpose:  Demonstrates how to solve the multi-processor
#              scheduling problem using the Fusion API.
##


import sys
import random
import mosek
from mosek.fusion import *
```

```
def main():
    #Parameters:
    n= 1000              #Number of tasks
    m= 8                 #Number of processors

    lb= 1.               #Range of short task lengths
    ub= 5.

    sh= 0.8              #Proportion of short tasks
    n_short= int(n*sh)
    n_long= n-n_short

    random.seed(0)
    T= sorted( [random.uniform(lb,ub) for i in range(n_short)]
              +[random.uniform(20*lb,20*ub) for i in range(n_long)], reverse=True)

    print("# jobs(n)    : ",n)
    print("# machine(m): ",m)

    with Model('Multi-processor scheduling') as M:

        x= M.variable('x', [m,n], Domain.binary())

        t= M.variable('t',1, Domain.unbounded())

        M.constraint( Expr.sum(x,0), Domain.equalsTo(1.) )

        M.constraint( Expr.sub( Var.repeat(t,m), Expr.mul(x,T) ) , Domain.greaterThan(0.) )

        M.objective( ObjectiveSense.Minimize, t )

        #LPT heuristic
        schedule=[0. for i in range(m)]
        init= [0. for i in range(n*m)]

        for i in range(n):
            mm= schedule.index(min(schedule))
            schedule[mm]+= T[i]
            init[n*mm+i]=1.

        #Comment this line to switch off feeding in the initial LPT solution
        x.setLevel(init)

        M.setLogHandler(sys.stdout)
        M.setSolverParam("mioTolRelGap", .01)
        M.solve()

        print('initial solution:')
        for i in range(m):
            print('M',i,init[i*n:(i+1)*n])

        print('MOSEK solution:')
        for i in range(m):
            print('M',i,[y for y in x.slice([i,0],[i+1,n]).level()])

if __name__ == '__main__':
    main()
```

7.9 Traveling Salesman Problem (TSP)

The *Traveling Salesman Problem* is one of the most famous and studied problem in combinatorics and integer optimization.

The main purpose of this case studies is to:

- show how to compactly define a model with *Fusion*;
- implement an iterative algorithm that solves a sequence of optimization problems;
- modify an optimization problem by additional constraints;
- accessing the solution of an optimization problem.

The material presented in this section draws inspiration from *[Pat03]*.

We are given a directed graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs. To each arc $(i, j) \in A$ corresponds a nonnegative cost c_{ij}. We aim to find a tour on G, i.e. a path touching all nodes only once, with minimum cost. For example let's consider the simple graph in Fig. 7.3.

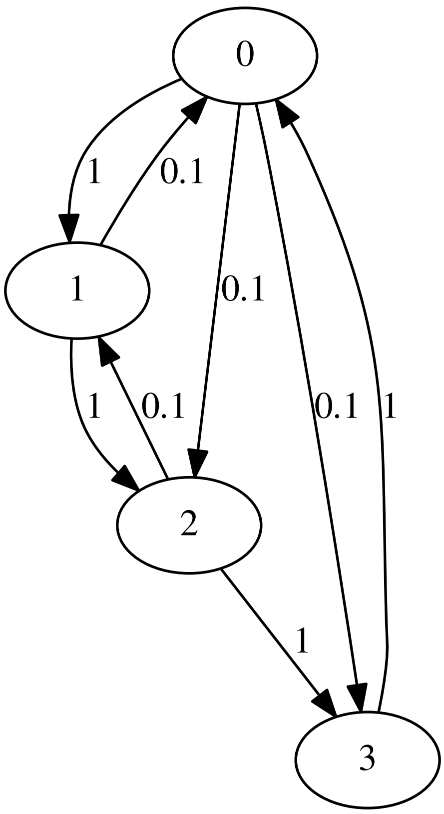


Fig. 7.3: A TSP example.

That corresponds to following adjacency and cost matrices A and c respectively:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad c = \begin{bmatrix} 0 & 1 & 0.1 & 0.1 \\ 0.1 & 0 & 1 & 0 \\ 0 & 0.1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Typically, the problem is modelled introducing a set of binary variables x_{ij} such that

$$x_{ij} = \begin{cases} 0 & \text{if arc } (i,j) \text{ is in the tour} \\ 1 & \text{otherwise.} \end{cases}$$

TSP is based on a simple assignement model:

$$\begin{array}{lll} \min & \sum_{i<j} c_{ij} x_{ij} & \\ & \sum_i x_{ij} = 1 & \forall j = 1,\ldots,n \\ & \sum_j x_{ij} = 1 & \forall i = 1,\ldots,n \\ & x_{ij} \in \{0,1\} & \forall i,j \end{array} \tag{7.29}$$

Problem (7.29) can be easily implemented in *Fusion*:

```
with Model() as M:

    x = M.variable([n,n], Domain.binary())

    M.constraint( Expr.sum(x,0), Domain.equalsTo(1.0))
    M.constraint( Expr.sum(x,1), Domain.equalsTo(1.0))

    M.objective(ObjectiveSense.Minimize, Expr.dot( C ,x)  )
```

Note in particular how:

- we can sum over rows and/or column using the *Expr.sum* function;
- we can use *Expr.dot* to compute the objective function.

Solving problem (7.29) will not yield a valid TSP, but only ensure the path will pass only once through each node. It can be shown that the solution of problem (7.29) actually is composed by a set of node disjoined sub-tours. In our example we get the solution depicted in Fig. 7.4.

i.e. there are two loops, namely *0->3->0* and *1->2->1*.

To obtain a solution to the TSP, we need some more constraints. One of the classical approach is the so-called *subtour elimination*: give a solution of (7.29) that is composed by at least two subtours, we add constraints that explicitly avoid that subtours:

$$\sum_{(i,j)\in c} x_{ij} \leq |c| - 1 \quad \forall c \in C \tag{7.30}$$

Thus the problem we want to solve at each step is

$$\begin{array}{lll} \min & \sum_{i<j} c_{ij} x_{ij} & \\ & \sum_i x_{ij} = 1 & \forall j = 1,\ldots,n \\ & \sum_j x_{ij} = 1 & \forall i = 1,\ldots,n \\ & x_{ij} \in \{0,1\} & \forall i,j \\ & \sum_{(i,j)\in c} x_{ij} \leq |c| - 1 & \forall c \in C \end{array} \tag{7.31}$$

The overall scheme is the following:

1. set C as the empty set
2. solve problem (7.31)
3. **if** x has only one cycle **stop**
4. **else** add cycles to C and **goto** 2

Cycle detection is a fairly easy task. We omit the procedure here for sake of simplicity. We only assume that as results we obtain a list of cycles C , each one listing the arcs it is composed by.

Now we need to add a constraint for for each cycle. Since we have the list of arcs, and each one corresponds to a variable *x[i][j]*, we can use the arc list and the function *Variable.pick* to define compactly constraints of the form (7.30)

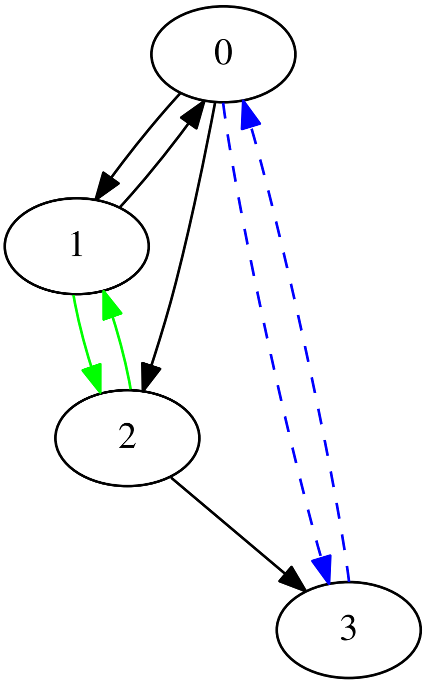


Fig. 7.4: Infeasible TSP solution: it has two disjoint loops.

```
for c in cycles:
    M.constraint( Expr.sum(x.pick(c)), Domain.lessThan( 1.0 * len(c) - 1 ) )
```

Executing our procedure will yield the following output

```
it #1 - solution cost: 2.200000

cycles:
[0,3] - [3,0] -
[1,2] - [2,1] -

it #2 - solution cost: 4.000000

cycles:
[0,1] - [1,2] - [2,3] - [3,0] -

solution:
 0  1  0  0
 0  0  1  0
 0  0  0  1
 1  0  0  0
```

Thus we first discover the two cycle solution we knew; then the second iteration is forced not to include those cycles, and a new solution is located. This time it is composed by one loop, and as expected the cost is higher. The solution is depicted in Fig. 7.5.

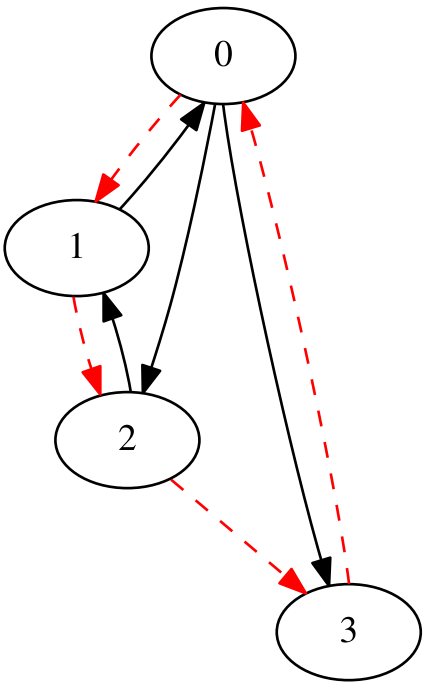


Fig. 7.5: The solution of the TSP example.

Formulation (7.31) can be improved in some cases exploiting the graph structure. Some simple tricks follow.

Self-loops

In case the graph contains self-loops, they are of no interest. Typically self-loop removal relies either on the definition of huge costs on that arcs or on the subtours elimination. Despite this works in practice, it is more advisable to just fix the corresponding variables to zero, i.e.

$$x_{ii} = 0 \quad \forall i = 1, \dots, n \tag{7.32}$$

These constraints will remove not only redundant variables, but also avoid unecessary large coefficients that can negatively affect the solver.

Constraints (7.32) are easily implemented as follows:

```
M.constraint( x.diag(), Domain.equalsTo(0.))
```

Two-arc loops removal

Assuming that we want to work on networks with more than two nodes, then it is reasonable to remove loops composed by only two arcs. This kind of loops are simple to define and come in reasonable number. The constraints we need to add are

$$x_{ij} + x_{ji} \leq 1 \quad \forall i, j = 1, \dots, n \tag{7.33}$$

Constraints (7.33) are easily implemented as follows:

```
M.constraint( Expr.add( x, x.transpose()), Domain.lessThan(1.0))
```

Forcing graph topology

In many application the graph is actually quite sparse, as for instance if it is a road network. For this reason many x_{ij} 's can be fixed to zero. Defining A as the adjacency matrix of G , then we can just force the following constraints

$$x_{ij} \leq A_{ij} \quad \forall i, j = 1, \dots, n \tag{7.34}$$

Constraints (7.34) translate in *Fusion* as:

```
M.constraint( x, Domain.lessThan( A ))
```

The complete working example

The complete code follows in Listing 7.18.

Listing 7.18: The complete code for the TSP examples.

```
##
#  Copyright:  Copyright (c) MOSEK ApS, Denmark. All rights reserved.

#
#  File:      tsp.py
#
#  Purpose: Demonstrates a simple technique to the TSP
#           usign the Fusion API.
#
##
```

```python
import mosek
from mosek.fusion import *
import sys


def tsp(n, A, C, graph_topology, remove_selfloops, remove_2_hop_loops):

    with Model() as M:

        x = M.variable([n,n], Domain.binary())

        M.constraint( Expr.sum(x,0), Domain.equalsTo(1.0))
        M.constraint( Expr.sum(x,1), Domain.equalsTo(1.0))

        M.objective(ObjectiveSense.Minimize, Expr.dot( C ,x)  )


        if remove_2_hop_loops:
            M.constraint( Expr.add( x, x.transpose()), Domain.lessThan(1.0))

        if graph_topology:
            M.constraint( x, Domain.lessThan( A ))

        if remove_selfloops:
            M.constraint( x.diag(), Domain.equalsTo(0.))

        it = 1

        while True:

            print("\n\n--------------------\nIteration",it)
            M.solve()

            print( '\nsolution cost:', M.primalObjValue())
            print( '\nsolution:')

            cycles = []

            for i in range(n):
                xi = x.slice([i,0],[i+1,n])
                print(xi.level())

                for j in range(n):
                    if xi.level()[j] <= 0.5 : continue

                    found = False
                    for c in cycles:
                        if len( [ a for a in c if i in a or j in a ] )> 0:
                            c.append( [i,j] )
                            found = True
                            break

                    if not found:
                        cycles.append([ [ i,j ]])

            print("\ncycles:")
            print([c for c in cycles])

            if len(cycles)==1:
                break;

            for c in cycles:
                M.constraint( Expr.sum(x.pick(c)), Domain.lessThan( 1.0 * len(c) - 1 ) )
```

```
            it = it +1

        return x.level(),c

    return [],[]

def main():
    A_i = [0,1,2,3,1,0,2,0]
    A_j = [1,2,3,0,0,2,1,3]
    C_v = [1.,1.,1.,1.,0.1,0.1,0.1,0.1]

    n= max(max(A_i),max(A_j))+1

    costs = Matrix.sparse(n,n,A_i,A_j,C_v)

    x,c = tsp(n, Matrix.sparse(n,n,A_i,A_j,1.), costs , True, True, True)

    x,c = tsp(n, Matrix.sparse(n,n,A_i,A_j,1.), costs , True, True, False)


if __name__ == '__main__':
    main()
```

CHAPTER

EIGHT

INTERACTION WITH THE SOLVER

The *Model* class is the interface to the solver for that specific model. When a *Model* class is instantiated, the solver environment is created.

Through the *Model* methods the user can retrieve information from the solver as:

- last *execution/solution status*

and *set options* about

- algorithm selection
- algorithm tolerances and stopping criteria
- *input/output format options* for the input/output operation.

The *Fusion* API provides the most commonly used solver functionality and options. If some advanced functionalities are needed, low level access t othe solver can be obtained from the model, see Section *8.4*.

Additional issues:

- how to stop the solver when key–pressed `CTRL+C` is not available is covered in Section *8.5*.
- it is possible to use call-back, thoght some limitations apply: see *8.6*.

Section index:

8.1 Solver Parameters

MOSEK comes with a large number of parameters that allows the user to tune its behavior. In *Fusion*, parameters can be set using the method *Model.setSolverParam*. Each parameter is identified by a unique string and accept either integers, floating point values or symbolic strings. *Fusion* tries to convert the value given by the user to the relevant type for the parameter.

If the conversion fails, an exception of type *FusionException* is thrown. Therefore it is always a good idea to incapsulate code setting parameters by an exception-catching block.

A complete reference of the parameters is available in Section *13.4*.

Real and Integer Parameters

These parameters can be specified both as a numerical type or a string. *Fusion* will cast the input to the desired type.

For instance a real parameter as *optimizerMaxTime*, the command

```
M.setSolverParam('optimizerMaxTime', 100.0)
```

would have the same effect as

```
M.setSolverParam('optimizerMaxTime', 100)
```

or

```
M.setSolverParam('optimizerMaxTime', '100.0')
```

On the other hand,

```
M.setSolverParam('optimizerMaxTime', '100 s.')
```

will fail throwing an exception of type *FusionException*.

String and Symbolic Parameters

These parameters accept strings, and therefore any other data type is not accepted.

Some parameters accept symbolic strings. For instance the parameter *presolveUse* accept a string among `on`, msk:const:*off*, `free`:

```
M.setSolverParam('presolveUse', 'off')
```

Any other string will not be accepted.

8.2 Problem and Solution Status

Once the solver terminates, it is time to check the results. The solver provides two different statuses that the user can inquery upon:

- *solution status*: information about the solution optimality degreee (optimal, nearly-optimal,...)
- *problem status*: information about the problem itseelf (feasibility, unboundness,...)

It is of the utmost importance to be able to fully understand the statuses that can be returned by the solver.

8.2.1 Solution Status

In principle, the only meaningful solution the user should care for is the optimal one. When it is not available the solver should have issued an infeasibility certificate. This behavior is clearly overoptimistic: for instance the solver might have been stopped by a time limit reached, or the execution stalled just before optimality had been reached. For this reason **MOSEK** actually distinguishes several solution statuses, some being

- **optimal** (*Optimal*)
- **near optimal** (*NearOptimal*)
- **unknown** (*Unknown*)

The complete list is available *SolutionStatus*. After **MOSEK** terminates, users should check the solution status using the functions *Model.getPrimalSolutionStatus* and *Model.getDualSolutionStatus*. Depending on that status the user can decide the action to be taken. Often a suboptimal solution is still valuable and deserve attention.

When a solution is recovered from a *Variable* object, it is only available if its status is among those considered *acceptable*. Otherwise an exception of type *SolutionError* is thrown.

It is therefore a good practice to

- protect the code against such exceptions

- investigate the reasons whenever they happens.

By default, acceptable status is *NearOptimal*. This can be changed using the function *Model.acceptedSolutionStatus*. For instance, if we want to accept every solution which is at least feasible we may write

```
M.acceptedSolutionStatus(AccSolutionStatus.Feasible)
```

while with

```
M.acceptedSolutionStatus(AccSolutionStatus.Anything)
```

we accept all available solutions.

Important: It is a user responsibility to check the actual solution quality.

To enquiry about the solution status accepted by a given *Model* instance just say

```
M.getAcceptedSolutionStatus()
```

8.2.2 Problem Status

The problem status is mainly concerned about whether the given optimization model is feasible. **MOSEK** is able to certified the infeasibility of conic problems up to a certain degree of numerical accuracy. The problem status can be checked using the *Model.getProblemStatus*.

Once the optimization terminates, it is good practice to inspect the results not only in terms of solution status, but also to check whether the problem has been certified feasible. In particular, if the solution status is not optimal, then the problem may be infeasible. To check for infeasibility we may write

```python

acceptable = [
    ProblemStatus.PrimalAndDualFeasible,
    ProblemStatus.PrimalFeasible,
    ProblemStatus.DualFeasible]

if M.getProblemStatus(SolutionType.Interior) in acceptable:
       #Here I should get the solution....
```

8.2.3 Accessing Solution Values

If a solution has been accepted, we can query for the objective function value for the primal and dual problems. They are readly available by the *Model.primalObjValue* and *Model.dualObjValue*, respectively.

Values attained by variables and constraints are available by the *Variable.level* method in classes *Variable* and *Constraint*, respectively: *Fusion* returns a flat array of values that the user can afterwards reshape.

In the same way users can access the corresponding dual values for variables and constraints, using the *Constraint.dual* method.

8.3 Input/Output

Through the *Model* class users can also control the solver I/O. This includes:

- *Execution logging*

- *Pretty printing*
- *Dump problem to disk.*

8.3.1 Logging

By default the solver runs silently and does not produce any output. In fact the output is discarded. However, the output of the solver can be redirected to any output stream using the method *Model.setLogHandler*. For instance, we can use the standard output

```
M.setLogHandler(sys.stdout)
```

A stream can be detached by passing NULL.

8.3.2 Pretty Printing

Fusion includes pretty printing for variables, matrices, expressions and constraints: a call to the method *toString()* returns a plain text representation of the object. This is particularly useful during development when one need to debug models and make sure that the model that has been defined is what it is meant to be.

In general, *Fusion* prints

- object type
- size and dimension
- a human readable representation

In general, a sparse representation of any object is printed whenever possible.

> **Warning:** Pretty printing of too large models is most likely unreadable!

Specific information follows.

fusion.Variable

A compact textual representation can be easily obtained: for instance a one dimensional variable called x will be printed just saying

```
n = 4
x = M.variable("x", n, Domain.greaterThan(0.))
print( x.toString() )
```

with the following output

```
LinearVariable( ('x',4) )
```

fusion.Matrix

Matrices are printed either in dense row-wise form or sparse triplet form. For instance, given a 2×4 matrix filled with ones, we can print it out as

```
print(Matrix.ones(2,4).toString())
```

producing the following output

```
DenseMatrix(2,4: [ 1.0,2.0,3.0,4.0 ],[ 5.0,6.0,7.0,8.0 ])
```

For a sparse matrix, for instance the identity

```
print(Matrix.eye(4).toString())
```

we get

```
SparseMatrix(4,4, [(0,0,1.0),(1,1,1.0),(2,2,1.0),(3,3,1.0) ])
```

fusion.Expression

Expressions are organized as matrices, and they share the overall layout. In particular, expressions are printed in sparse format. For instance

```
x = M.variable("x", 4 , Domain.unbounded());
ee = Expr.mul(Matrix.eye(4), x);
print( ee.toString())
```

It will produce the following output

```
Expr(ndim=(4),
     [  + 1.0 x[0],
        + 1.0 x[1],
        + 1.0 x[2],
        + 1.0 x[3] ])
```

In this case the expression is stored as a one dimensional array. The following case shows what happens with sparsity: we multipy element-wise the identity matrix times a bi-dimensional squared variable, i.e.

```
x = M.variable("x", [4,4], Domain.unbounded())
ee = Expr.mulElm(Matrix.eye(4), x)
print( ee.toString() )
```

It will produce the following output

```
Expr(ndim=(4,4),
     [ ([0 0]) ->  + 1.0 X[0,0],
       ([1 1]) ->  + 1.0 X[1,1],
       ([2 2]) ->  + 1.0 X[2,2],
       ([3 3]) ->  + 1.0 X[3,3] ])
```

As expected the result is a squared matrix of the same dimension, but only the non zeros entries are printed.

fusion.Constraint

A compact representation a the constraint can be obtain using the *Constraint.toString*. For instance a set of linear constraints of the form $Ix = 0$, with I being the identity matrix is implemented can be visualize as

```
n = 4
x = M.variable('x',n, Domain.greaterThan(0.))
c = M.constraint('c', Expr.mul(Matrix.eye(n),x), Domain.equalsTo(0.))

print(c.toString())
```

The output is

```
Constraint( 'c', (4),
  c[0] :   + 1.0 x[0]   = 0.0,
  c[1] :   + 1.0 x[1]   = 0.0,
  c[2] :   + 1.0 x[2]   = 0.0,
  c[3] :   + 1.0 x[3]   = 0.0 )
```

Notice that only non zero entries are printed.

The printed representation also includes all auxiliary variables introduced by *Fusion*. For instance a single second order cone of the form

8.3.3 Dumping a Problem to File

A model can be dump to file using the *Model.writeTask*, just specifying the file name. The file type will be deduced automatically by the extension. For instance

```
M.writeTask('dump.mps')
```

will dump the model to an MPS file. Supported formats are listed in Section *14*.

All format can be gzipped appending the *.gz* extension, i.e. the command

```
M.writeTask('dump.mps.gz')
```

will produce an MPS file compressed in gzipped format.

> **Warning:** The dumped model also contains all the additional variables generated when defining cones.

It is therefore advisable to assign meaningful names to variables when debugging, in order to improved readibility.

For more details please refer to Section *14*.

8.4 Access to Optimizer API Task

The *Model* class acts as a tiny wrapper on top of a **MOSEK** task. Some low-level functionalities provided by the task in the optimizer API are not directly supported by *Fusion*. Instead, the task handler can be obtained by the method *Model.getTask*.

> **Warning:** The task handler is **not** a copy and any modification may invalidate or corrupt the model.

Therefore the access to the task should be considered carefully and avoided unless special functionalities are required.

8.5 Stopping the Solver Execution

To force **MOSEK** to stop, *Fusion* class *Model* provides the method *Model.breakSolver* that notifies the solver that it must stop as soon as possible. The solver periodically test for such notification and as it happens, it will stop the execution. The state of the solver and solution may be undefined (see Section *8.2*).

Note: The built-in stopping criteria should be used instead whenever possible!

The typical steps are the following:

1. build the optimization model (say `M`) as usual;
2. create a separate task in which `M` will run;
3. once the termination criterion is met call the function *Model.breakSolver* on `M`.

Warning: These steps are very language dependent and particular care must be taken to avoid stalling or other undesired side effects.

8.5.1 A Working Example: Setting a Time Limit

In this example we will use a time limit as an additional stopping criterion, despite the fact that a time limit is available as a parameter in **MOSEK**.

We will use a simple MIP model which we know it runs for quite long time

$$
\begin{array}{lll}
\min & \sum_i x_i & \\
s.t. & \sum_{i \in P_j} x_i = 1 & j = 1, \ldots, m \\
 & x_i \in \{0, 1\} & j = 1, \ldots, n
\end{array}
$$

where P_j is a permutation of $\{1, \ldots, n\}$ such that $|P_j| = p$. This model is declared as

```
with Model('SolveBinary') as M:
    M.setLogHandler(sys.stdout)

    x = M.variable("x", n, Domain.binary())

    M.objective(ObjectiveSense.Minimize, Expr.sum(x))
    M.setLogHandler(sys.stdout)

    L = list(range(n))
    for i in range(m):
        R.shuffle(L)
        M.constraint(Expr.sum(x.pick(L[:p])),Domain.equalsTo(p // 2))
```

Once the model has been built, we proceed creating a new thread that will be responsible for the actual solver execution:

```
T = threading.Thread(target = M.solve)
```

Then in the main thread we can check for the criterion to be satisfied

- a time limit of five seconds
- the user pressing `CTRL+C`

It must be notice that we need to ensure that the execution on the main threads resumes after the solver actually terminates, i.e. the auxiliary threads returns. This is performed by the following lines:

```
    # Loop until we get a solution or you run out of patience and press Ctrl-C
    while True:
        if not T.isAlive():
            print("Solver terminated before anything happened!")
            break
        elif time.time()-T0 > timeout:
            print("Solver terminated due to timeout!")
```

```
            M.breakSolver()
            break
except KeyboardInterrupt:
    print("Signalling the solver that it can give up now!")
    M.breakSolver()
```

The complete source code follows in Listing 8.1 .

Listing 8.1: Example on how stop solver execution.

```
import sys
import mosek.fusion
from   mosek.fusion import *
import random
import threading
import time

def main():

    timeout = 5

    n = 200    # number of binary variables
    m = n // 5 # number of constraints
    p = n // 5 # Each constraint picks p variables and requires that exactly half of them are 1

    R = random.Random(1234)


    print("Build problem...")
    with Model('SolveBinary') as M:
        M.setLogHandler(sys.stdout)

        x = M.variable("x", n, Domain.binary())

        M.objective(ObjectiveSense.Minimize, Expr.sum(x))
        M.setLogHandler(sys.stdout)

        L = list(range(n))
        for i in range(m):
            R.shuffle(L)
            M.constraint(Expr.sum(x.pick(L[:p])),Domain.equalsTo(p // 2))
        print("Start thread...")
        T = threading.Thread(target = M.solve)
        T0 = time.time()

        try:
            T.start() # optimization now running in background

            # Loop until we get a solution or you run out of patience and press Ctrl-C
            while True:
                if not T.isAlive():
                    print("Solver terminated before anything happened!")
                    break
                elif time.time()-T0 > timeout:
                    print("Solver terminated due to timeout!")
                    M.breakSolver()
                    break
        except KeyboardInterrupt:
            print("Signalling the solver that it can give up now!")
            M.breakSolver()
        finally:
            try:
                T.join() # wait for the solver to return
```

```
            except:
                pass
if __name__ == '__main__':
  main()
```

8.6 Callbacks in *Fusion*

Callbacks are a very useful mechanism to interact with the solver at runtime and modify its behaviour. *Fusion* provides a limited support for callback that allows the user to hook a function to the solver progress callback. This entry point is regularly called during the optimization and can be used to

- obtain a customized log of the solver execution,
- collect information for debugging purpose or
- ask the solver to terminate.

Important: The only function that can be called from the callback is the *Model.breakSolver*, indicating that the solver should terminate. No other functions *must* be called from the callback. Otherwise the execution state of the solver and its outcome are undefined.

The callback can be set calling the *Model.setCallbackHandler* method.

8.6.1 A minimal Working Example

The following example is based on the basic tutorial in Section *4.1*. It is extended providing the solver with a callback funtion that print out additional output during the optimization algorithm execution. The output depends on the eleted algorithm (primal/dual simplex or innterio-point). Moreover, the execution is terminated when a given time limit is reached.

```
def makeUserCallback(maxtime):
    def userCallback(caller,
                     douinf,
                     intinf,
                     lintinf):
        opttime = 0.0
        if caller == callbackcode.begin_intpnt:
            print ("Starting interior-point optimizer")
        elif caller == callbackcode.intpnt:
            itrn    = intinf[iinfitem.intpnt_iter      ]
            pobj    = douinf[dinfitem.intpnt_primal_obj]
            dobj    = douinf[dinfitem.intpnt_dual_obj  ]
            stime   = douinf[dinfitem.intpnt_time      ]
            opttime = douinf[dinfitem.optimizer_time   ]

            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f) " % (opttime,stime))
            print ("  Primal obj.: %-18.6e  Dual obj.: %-18.6e" % (pobj,dobj))
        elif caller == callbackcode.end_intpnt:
            print ("Interior-point optimizer finished.")
        elif caller == callbackcode.begin_primal_simplex:
            print ("Primal simplex optimizer started.")
        elif caller == callbackcode.update_primal_simplex:
            itrn    = intinf[iinfitem.sim_primal_iter  ]
            pobj    = douinf[dinfitem.sim_obj          ]
            stime   = douinf[dinfitem.sim_time         ]
            opttime = douinf[dinfitem.optimizer_time   ]
```

```
            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f)" % (opttime,stime))
            print ("  Obj.: %-18.6e" % pobj )
        elif caller == callbackcode.end_primal_simplex:
            print ("Primal simplex optimizer finished.")
        elif caller == callbackcode.begin_dual_simplex:
            print ("Dual simplex optimizer started.")
        elif caller == callbackcode.update_dual_simplex:
            itrn    = intinf[iinfitem.sim_dual_iter    ]
            pobj    = douinf[dinfitem.sim_obj          ]
            stime   = douinf[dinfitem.sim_time         ]
            opttime = douinf[dinfitem.optimizer_time   ]
            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f)" % (opttime,stime))
            print ("  Obj.: %-18.6e" % pobj)
        elif caller == callbackcode.end_dual_simplex:
            print ("Dual simplex optimizer finished.")
        elif caller == callbackcode.begin_bi:
            print ("Basis identification started.")
        elif caller == callbackcode.end_bi:
            print ("Basis identification finished.")
        else:
            pass
        if opttime >= maxtime:
            # mosek is spending too much time. Terminate it.
            return 1

        return 0

    return userCallback
```

The callback function is then simply hooked:

```
usrcallback =  makeUserCallback(maxtime = 3600)
M.setCallbackHandler( usrcallback )
```

The complete working example follows in Listing 8.2.

Listing 8.2: Example of callback function with *Fusion*.

```
##
#   Copyright:  Copyright (c) MOSEK ApS, Denmark. All rights reserved.

#   File:       callback.py
#
#   Purpose:    To demonstrate how to use the progress
#               callback.
#
#
#               callback psim
#               callback dsim
#               callback intpnt
#               The first argument tells which optimizer to use
#               i.e. psim is primal simplex, dsim is dual simplex
#               and intpnt is interior-point.
##
import sys
from mosek.fusion import *

def makeUserCallback(maxtime):
    def userCallback(caller,
                     douinf,
                     intinf,
```

```
                          lintinf):
        opttime = 0.0
        if caller == callbackcode.begin_intpnt:
            print ("Starting interior-point optimizer")
        elif caller == callbackcode.intpnt:
            itrn    = intinf[iinfitem.intpnt_iter      ]
            pobj    = douinf[dinfitem.intpnt_primal_obj]
            dobj    = douinf[dinfitem.intpnt_dual_obj  ]
            stime   = douinf[dinfitem.intpnt_time      ]
            opttime = douinf[dinfitem.optimizer_time   ]

            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f) " % (opttime,stime))
            print ("  Primal obj.: %-18.6e  Dual obj.: %-18.6e" % (pobj,dobj))
        elif caller == callbackcode.end_intpnt:
            print ("Interior-point optimizer finished.")
        elif caller == callbackcode.begin_primal_simplex:
            print ("Primal simplex optimizer started.")
        elif caller == callbackcode.update_primal_simplex:
            itrn    = intinf[iinfitem.sim_primal_iter  ]
            pobj    = douinf[dinfitem.sim_obj          ]
            stime   = douinf[dinfitem.sim_time         ]
            opttime = douinf[dinfitem.optimizer_time   ]

            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f)" % (opttime,stime))
            print ("  Obj.: %-18.6e" % pobj )
        elif caller == callbackcode.end_primal_simplex:
            print ("Primal simplex optimizer finished.")
        elif caller == callbackcode.begin_dual_simplex:
            print ("Dual simplex optimizer started.")
        elif caller == callbackcode.update_dual_simplex:
            itrn    = intinf[iinfitem.sim_dual_iter    ]
            pobj    = douinf[dinfitem.sim_obj          ]
            stime   = douinf[dinfitem.sim_time         ]
            opttime = douinf[dinfitem.optimizer_time   ]
            print ("Iterations: %-3d" % itrn)
            print ("  Elapsed time: %6.2f(%.2f)" % (opttime,stime))
            print ("  Obj.: %-18.6e" % pobj)
        elif caller == callbackcode.end_dual_simplex:
            print ("Dual simplex optimizer finished.")
        elif caller == callbackcode.begin_bi:
            print ("Basis identification started.")
        elif caller == callbackcode.end_bi:
            print ("Basis identification finished.")
        else:
            pass
        if opttime >= maxtime:
            # mosek is spending too much time. Terminate it.
            return 1

        return 0

    return userCallback

def main(slvr):

    if slvr not in ['psim','dsim','intpnt']: return

    A = [ [ 3.0, 2.0, 0.0, 1.0 ],
          [ 2.0, 3.0, 1.0, 1.0 ],
          [ 0.0, 0.0, 3.0, 2.0 ] ]
    c = [ 3.0, 5.0, 1.0, 1.0  ]
```

```
    with Model() as M:
        x = M.variable("x", 3, Domain.greaterThan(0.0))
        y = M.variable("y", 1, Domain.inRange(0.0, 10.0))
        z = Var.vstack(x,y)
        M.constraint("c1", Expr.dot(A[0], z), Domain.equalsTo(30.0))
        M.constraint("c2", Expr.dot(A[1], z), Domain.greaterThan(15.0))
        M.constraint("c3", Expr.dot(A[2], z), Domain.lessThan(25.0))
        M.objective("obj", ObjectiveSense.Maximize, Expr.dot(c, z))

        if slvr == 'psim':
            M.setSolverParam('optimizer', 'primal_simplex')
        elif slvr == 'dsim':
            M.setSolverParam('optimizer', 'dual_simplex')
        else:
            M.setSolverParam('optimizer', 'intpnt')

        usrcallback =  makeUserCallback(maxtime = 3600)
        M.setCallbackHandler( usrcallback )
        M.setSolverParam('log', 0)

        M.solve()



if __name__ == '__main__':
    main(sys.argv[1] if len(sys.argv)>1 else 'intpnt')
```

CHAPTER

NINE

PERFORMANCE CONSIDERATIONS

9.1 Sparse Matrices

Deciding whether a matrix should be stored in dense or sparse format is not always trivial and it does not only depend on storage considerations. For a given $n \times m$ matrix with l non zero entries, the storage required is proportional to

- $n \cdot m$ for a dense matrix,
- $3 \cdot l$ for a sparse matrix.

Therefore if $l << n \cdot m$, then the required storage in sparse form is much smaller than in dense format. The consequences are

- reduced memory requirements,
- faster expression computation,
- meet the internal solver representation.

However, there are borderline cases in which these advantages may vanish due to overhaed on creating the triplets representation.

Sparsity is a key feature of many optimization models and happens often naturally. For instance, linear constraints arising from networks or multi-period planning are typically sparse. *Fusion* does not detect sparsity but leaves to the user the responsibility to choose the msot appropriate storage format. It provides adaptors for sparse matrices by *Matrix* static methods such as *Matrix.sparse* or *Matrix.diag*.

9.2 Nested Expressions

A possible source of performance degradation is an excessive use of nested expressions. For example

$$\sum_{i=1}^{n} A_i x_i$$

$$x_i \in \mathbb{R}^k, A_i \in \mathbb{R}^{k \times k},$$

it could be expressed in a loop as

```
ee = Expr.constTerm(k, 0.)
for i in range(n):
   ee = Expr.add( ee, Expr.mul(A[i],x[i] ) )
```

A better way is to store intermediate expressions for $A_i x_i$ and sum all of them in one step:

```
Exp.add( [ Expr.mul(AA,xx) for (AA,xx) in zip(AA,xx)] )
```

This implementation is more efficient as it reduces the number of intermediate expressions.

9.3 Names

Fusion makes very easy to specify names for variables, constraints and the objective function. It is very useful for debugging and improves the readability of problems stored in files. But unfortunately it comes at a price:

- *Fusion* must check and make sure that names are unique

To reduce the overhead, names are actually generated when some operation explicitly ask for them. For example, if we want to print a variable information with the following code

```
x = m.variable("x", 10, Domain.unbounded())

print(x.toString())
```

with the following output

Fusion generates unique names for the x entries when *Variable.toString* is called.

To optimize performances it is therefore advisable to not specify names at all. Notice that a careful choice of variable names makes the code very readable with no needs for labels.

CHAPTER

TEN

PROBLEM FORMULATION AND SOLUTIONS

In this chapter we will discuss the following issues:

- The formal definitions of the problem types that **MOSEK** can solve.
- The solution information produced by **MOSEK**.
- The information produced by **MOSEK** if the problem is infeasible.

10.1 Linear Optimization

A linear optimization problem can be written as

$$
\begin{array}{lccccc}
\text{minimize} & & & c^T x + c^f & & \\
\text{subject to} & l^c & \leq & Ax & \leq & u^c, \\
 & l^x & \leq & x & \leq & u^x,
\end{array}
\tag{10.1}
$$

where

- m is the number of constraints.
- n is the number of decision variables.
- $x \in \mathbb{R}^n$ is a vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear part of the objective function.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.

A primal solution (x) is *(primal) feasible* if it satisfies all constraints in (10.1). If (10.1) has at least one primal feasible solution, then (10.1) is said to be (primal) feasible.

In case (10.1) does not have a feasible solution, the problem is said to be *(primal) infeasible*

10.1.1 Duality for Linear Optimization

Corresponding to the primal problem (10.1), there is a dual problem

$$
\begin{array}{lccc}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f & & \\
 & A^T y + s_l^x - s_u^x & = & c, \\
\text{subject to} & -y + s_l^c - s_u^c & = & 0, \\
 & s_l^c, s_u^c, s_l^x, s_u^x & \geq & 0.
\end{array}
\tag{10.2}
$$

If a bound in the primal problem is plus or minus infinity, the corresponding dual variable is fixed at 0, and we use the convention that the product of the bound value and the corresponding dual variable is 0. E.g.

$$l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0 \text{ and } l_j^x \cdot (s_l^x)_j = 0.$$

This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. A solution

$$(y, s_l^c, s_u^c, s_l^x, s_u^x)$$

to the dual problem is feasible if it satisfies all the constraints in (10.2). If (10.2) has at least one feasible solution, then (10.2) is *(dual) feasible*, otherwise the problem is *(dual) infeasible*.

A Primal-dual Feasible Solution

A solution

$$(x, y, s_l^c, s_u^c, s_l^x, s_u^x)$$

is denoted a *primal-dual feasible solution*, if (x) is a solution to the primal problem (10.1) and $(y, s_l^c, s_u^c, s_l^x, s_u^x)$ is a solution to the corresponding dual problem (10.2).

The Duality Gap

Let

$$(x^*, y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$$

be a primal-dual feasible solution, and let

$$(x^c)^* := Ax^*.$$

For a primal-dual feasible solution we define the *duality gap* as the difference between the primal and the dual objective value,

$$\begin{array}{l} c^T x^* + c^f - \left\{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \right\} \\ \quad = \sum_{i=0}^{m-1} \left[(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*) \right] \\ \quad + \sum_{j=0}^{n-1} \left[(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*) \right] \geq 0 \end{array} \tag{10.3}$$

where the first relation can be obtained by transposing and multiplying the dual constraints (10.2) by x^* and $(x^c)^*$ respectively, and the second relation comes from the fact that each term in each sum is nonnegative. It follows that the primal objective will always be greater than or equal to the dual objective.

An Optimal Solution

It is well-known that a linear optimization problem has an optimal solution if and only if there exist feasible primal and dual solutions so that the duality gap is zero, or, equivalently, that the *complementarity conditions*

$$\begin{array}{rcll} (s_l^c)_i^* ((x_i^c)^* - l_i^c) & = & 0, & i = 0, \ldots, m-1, \\ (s_u^c)_i^* (u_i^c - (x_i^c)^*) & = & 0, & i = 0, \ldots, m-1, \\ (s_l^x)_j^* (x_j^* - l_j^x) & = & 0, & j = 0, \ldots, n-1, \\ (s_u^x)_j^* (u_j^x - x_j^*) & = & 0, & j = 0, \ldots, n-1, \end{array}$$

are satisfied.

If (10.1) has an optimal solution and **MOSEK** solves the problem successfully, both the primal and dual solution are reported, including a status indicating the exact state of the solution.

10.1.2 Infeasibility for Linear Optimization

Primal Infeasible Problems

If the problem (10.1) is infeasible (has no feasible solution), **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is the certificate of infeasibility, and the primal solution is undefined.

A certificate of primal infeasibility is a feasible solution to the modified dual problem

$$\begin{array}{ll}\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\ \text{subject to} & \\ & \begin{array}{r} A^T y + s_l^x - s_u^x = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \end{array}\end{array} \tag{10.4}$$

such that the objective value is strictly positive, i.e. a solution

$$(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$$

to (10.4) so that

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* > 0.$$

Such a solution implies that (10.4) is unbounded, and that its dual is infeasible. As the constraints to the dual of (10.4) are identical to the constraints of problem (10.1), we thus have that problem (10.1) is also infeasible.

Dual Infeasible Problems

If the problem (10.2) is infeasible (has no feasible solution), **MOSEK** will report a certificate of dual infeasibility: The primal solution reported is the certificate of infeasibility, and the dual solution is undefined.

A certificate of dual infeasibility is a feasible solution to the modified primal problem

$$\begin{array}{lccccc}\text{minimize} & & & c^T x & & \\ \text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\ & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \end{array} \tag{10.5}$$

where

$$\hat{l}_i^c = \left\{ \begin{array}{ll} 0 & \text{if } l_i^c > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_i^c := \left\{ \begin{array}{ll} 0 & \text{if } u_i^c < \infty, \\ \infty & \text{otherwise,} \end{array} \right\}$$

and

$$\hat{l}_j^x = \left\{ \begin{array}{ll} 0 & \text{if } l_j^x > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_j^x := \left\{ \begin{array}{ll} 0 & \text{if } u_j^x < \infty, \\ \infty & \text{otherwise,} \end{array} \right\}$$

such that

$$c^T x < 0.$$

Such a solution implies that (10.5) is unbounded, and that its dual is infeasible. As the constraints to the dual of (10.5) are identical to the constraints of problem (10.2), we thus have that problem (10.2) is also infeasible.

Primal and Dual Infeasible Case

In case that both the primal problem (10.1) and the dual problem (10.2) are infeasible, **MOSEK** will report only one of the two possible certificates — which one is not defined (**MOSEK** returns the first certificate found).

Minimalization vs. Maximalization

When the objective sense of problem (10.1) is maximization, i.e.

$$\begin{array}{lccccc} \text{maximize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \end{array}$$

the objective sense of the dual problem changes to minimization, and the domain of all dual variables changes sign in comparison to (10.2). The dual problem thus takes the form

$$\begin{array}{ll} \text{minimize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ \text{subject to} & \\ & A^T y + s_l^x - s_u^x = c, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \leq 0. \end{array}$$

This means that the duality gap, defined in (10.3) as the primal minus the dual objective value, becomes nonpositive. It follows that the dual objective will always be greater than or equal to the primal objective. The primal infeasibility certificate will be reported by **MOSEK** as a solution to the system

$$\begin{array}{c} A^T y + s_l^x - s_u^x = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \end{array} \tag{10.6}$$

such that the objective value is strictly negative

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* < 0.$$

Similarly, the certificate of dual infeasibility is an x satisfying the requirements of (10.5) such that $c^T x > 0$.

10.2 Conic Quadratic Optimization

Conic quadratic optimization is an extension of linear optimization (see Section *10.1*) allowing conic domains to be specified for subsets of the problem variables. A conic quadratic optimization problem can be written as

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & x \in \mathcal{K}, & & \end{array} \tag{10.7}$$

where set $\mathcal{K}$ is a Cartesian product of convex cones, namely $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_p$. Having the domain restriction, $x \in \mathcal{K}$, is thus equivalent to

$$x^t \in \mathcal{K}_t \subseteq \mathbb{R}^{n_t},$$

where $x = (x^1, \ldots, x^p)$ is a partition of the problem variables. Please note that the n-dimensional Euclidean space $\mathbb{R}^n$ is a cone itself, so simple linear variables are still allowed.

MOSEK supports only a limited number of cones, specifically:

- The $\mathbb{R}^n$ set.
- The quadratic cone:

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n : x_1 \geq \sqrt{\sum_{j=2}^{n} x_j^2} \right\}.$$

- The rotated quadratic cone:

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n : 2x_1x_2 \geq \sum_{j=3}^{n} x_j^2, \quad x_1 \geq 0, \quad x_2 \geq 0 \right\}.$$

Although these cones may seem to provide only limited expressive power they can be used to model a wide range of problems as demonstrated in *[MOSEKApS12]*.

10.2.1 Duality for Conic Quadratic Optimization

The dual problem corresponding to the conic quadratic optimization problem (10.7) is given by

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ \text{subject to} & \\ & A^T y + s_l^x - s_u^x + s_n^x = c \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ & s_n^x \in \mathcal{K}^*, \end{array} \tag{10.8}$$

where the dual cone $\mathcal{K}^*$ is a Cartesian product of the cones

$$\mathcal{K}^* = \mathcal{K}_1^* \times \cdots \times \mathcal{K}_p^*,$$

where each $\mathcal{K}_t^*$ is the dual cone of $\mathcal{K}_t$. For the cone types **MOSEK** can handle, the relation between the primal and dual cone is given as follows:

- The $\mathbb{R}^n$ set:

$$\mathcal{K}_t = \mathbb{R}^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \{ s \in \mathbb{R}^{n_t} : \quad s = 0 \}.$$

- The quadratic cone:

$$\mathcal{K}_t = \mathcal{Q}^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \mathcal{Q}^{n_t} = \left\{ s \in \mathbb{R}^{n_t} : s_1 \geq \sqrt{\sum_{j=2}^{n_t} s_j^2} \right\}.$$

- The rotated quadratic cone:

$$\mathcal{K}_t = \mathcal{Q}_r^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \mathcal{Q}_r^{n_t} = \left\{ s \in \mathbb{R}^{n_t} : 2s_1s_2 \geq \sum_{j=3}^{n_t} s_j^2, \quad s_1 \geq 0, \quad s_2 \geq 0 \right\}.$$

Please note that the dual problem of the dual problem is identical to the original primal problem.

10.2.2 Infeasibility for Conic Quadratic Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of infeasibility. This works exactly as for linear problems (see Section *10.1.2*).

Primal Infeasible Problems

If the problem (10.7) is infeasible, **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is the certificate of infeasibility, and the primal solution is undefined.

A certificate of primal infeasibility is a feasible solution to the problem

$$\begin{array}{lrcl} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x & & \\ \text{subject to} & & & \\ & A^T y + s_l^x - s_u^x + s_n^x & = & 0, \\ & -y + s_l^c - s_u^c & = & 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x & \geq & 0, \\ & s_n^x \in \mathcal{K}^*, & & \end{array}$$

such that the objective value is strictly positive.

Dual infeasible problems

If the problem (10.8) is infeasible, **MOSEK** will report a certificate of dual infeasibility: The primal solution reported is the certificate of infeasibility, and the dual solution is undefined.

A certificate of dual infeasibility is a feasible solution to the problem

$$\begin{array}{lccccc} \text{minimize} & & & c^T x & & \\ \text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\ & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \\ & & & x \in \mathcal{K}, & & \end{array}$$

where

$$\hat{l}_i^c = \left\{ \begin{array}{ll} 0 & \text{if } l_i^c > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_i^c := \left\{ \begin{array}{ll} 0 & \text{if } u_i^c < \infty, \\ \infty & \text{otherwise,} \end{array} \right\}$$

and

$$\hat{l}_j^x = \left\{ \begin{array}{ll} 0 & \text{if } l_j^x > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_j^x := \left\{ \begin{array}{ll} 0 & \text{if } u_j^x < \infty, \\ \infty & \text{otherwise,} \end{array} \right\}$$

such that the objective value is strictly negative.

10.3 Semidefinite Optimization

Semidefinite optimization is an extension of conic quadratic optimization (see Section *10.2*) allowing positive semidefinite matrix variables to be used in addition to the usual scalar variables. A semidefinite optimization problem can be written as

$$\begin{array}{lccccll} \text{minimize} & & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle + c^f & & & \\ \text{subject to} & l_i^c & \leq & \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \leq & u_i^c, & i = 0, \ldots, m-1 \\ & l_j^x & \leq & x_j & \leq & u_j^x, & j = 0, \ldots, n-1 \\ & & & x \in \mathcal{K}, \overline{X}_j \in \mathcal{S}_+^{r_j}, & & & j = 0, \ldots, p-1 \end{array} \tag{10.9}$$

where the problem has p symmetric positive semidefinite variables $\overline{X}_j \in \mathcal{S}_+^{r_j}$ of dimension r_j with symmetric coefficient matrices $\overline{C}_j \in \mathcal{S}^{r_j}$ and $\overline{A}_{i,j} \in \mathcal{S}^{r_j}$. We use standard notation for the matrix inner product, i.e., for $U, V \in \mathbb{R}^{m \times n}$ we have

$$\langle U, V \rangle := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} U_{ij} V_{ij}.$$

With semidefinite optimization we can model a wide range of problems as demonstrated in *[MOSEKApS12]*.

10.3.1 Duality for Semidefinite Optimization

The dual problem corresponding to the semidefinite optimization problem (10.9) is given by

$$\begin{array}{lll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f & \\ \text{subject to} & & \\ & c - A^T y + s_u^x - s_l^x = s_n^x, & \\ & \overline{C}_j - \sum_{i=0}^{m} y_i \overline{A}_{ij} = \overline{S}_j, & j = 0, \ldots, p-1 \\ & s_l^c - s_u^c = y, & \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ & s_n^x \in \mathcal{K}^*, \quad \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 0, \ldots, p-1 \end{array} \tag{10.10}$$

where $A \in \mathbb{R}^{m\times n}$, $A_{ij} = a_{ij}$, which is similar to the dual problem for conic quadratic optimization (see Section *10.2.1*), except for the addition of dual constraints

$$\left(\overline{C}_j - \sum_{i=0}^{m} y_i \overline{A}_{ij}\right) \in \mathcal{S}_+^{r_j}.$$

Note that the dual of the dual problem is identical to the original primal problem.

10.3.2 Infeasibility for Semidefinite Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of the infeasibility. This works exactly as for linear problems (see Section *10.1.2*).

Primal Infeasible Problems

If the problem (10.9) is infeasible, **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is a certificate of infeasibility, and the primal solution is undefined.

A certificate of primal infeasibility is a feasible solution to the problem

$$\begin{array}{lll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x & \\
\text{subject to} & & \\
 & A^T y + s_l^x - s_u^x + s_n^x = 0, & \\
 & \sum_{i=0}^{m-1} y_i \overline{A}_{ij} + \overline{S}_j = 0, & j = 0, \ldots, p-1 \\
 & -y + s_l^c - s_u^c = 0, & \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\
 & s_n^x \in \mathcal{K}^*, \quad \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 0, \ldots, p-1
\end{array}$$

such that the objective value is strictly positive.

Dual Infeasible Problems

If the problem (10.10) is infeasible, **MOSEK** will report a certificate of dual infeasibility: The primal solution reported is the certificate of infeasibility, and the dual solution is undefined.

A certificate of dual infeasibility is a feasible solution to the problem

$$\begin{array}{llclll}
\text{minimize} & & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle & & \\
\text{subject to} & \hat{l}_i^c & \leq & \sum_{j=1}^{n} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \leq & \hat{u}_i^c, \quad i = 0, \ldots, m-1 \\
 & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \\
 & & & x \in \mathcal{K}, \quad \overline{X}_j \in \mathcal{S}_+^{r_j}, & & \quad j = 0, \ldots, p-1
\end{array}$$

where

$$\hat{l}_i^c = \left\{ \begin{array}{ll} 0 & \text{if } l_i^c >; -\infty, \\ -\infty & \text{otherwise,} \end{array} \right. \quad \text{and} \quad \hat{u}_i^c := \left\{ \begin{array}{ll} 0 & \text{if } u_i^c <; \infty, \\ \infty & \text{otherwise,} \end{array} \right.$$

and

$$\hat{l}_j^x = \left\{ \begin{array}{ll} 0 & \text{if } l_j^x >; -\infty, \\ -\infty & \text{otherwise,} \end{array} \right. \quad \text{and} \quad \hat{u}_j^x := \left\{ \begin{array}{ll} 0 & \text{if } u_j^x <; \infty, \\ \infty & \text{otherwise,} \end{array} \right.$$

such that the objective value is strictly negative.

CHAPTER

ELEVEN

THE OPTIMIZERS FOR CONTINUOUS PROBLEMS

The most essential part of **MOSEK** is the optimizers. Each optimizer is designed to solve a particular class of problems, i.e. linear, conic, or general nonlinear problems. The purpose of the present chapter is to discuss which optimizers are available for the continuous problem classes and how the performance of an optimizer can be tuned, if needed. This chapter deals with the optimizers for *continuous problems* with no integer variables.

When the optimizer is called, it roughly performs the following steps:

1. *Presolve*: Preprocessing to reduce the size of the problem.
2. *Dualizer*: Choosing whether to solve the primal or the dual form of the problem.
3. *Scaling*: Scaling the problem for better numerical stability.
4. *Optimize*: Solve the problem using selected method.

The first three preprocessing steps are transparent to the user, but useful to know about for tuning purposes. In general, the purpose of the preprocessing steps is to make the actual optimization more efficient and robust.

Using multiple threads

The interior-point optimizers in **MOSEK** have been parallelized. This means that if you solve linear, quadratic, conic, or general convex optimization problem using the interior-point optimizer, you can take advantage of multiple CPU's.

By default **MOSEK** will automatically select the number of threads to be employed when solving the problem. However, the number of threads employed can be changed by setting the parameter `numThreads`. This should never exceed the number of cores on the computer.

The speed-up obtained when using multiple threads is highly problem and hardware dependent, and consequently, it is advisable to compare single threaded and multi threaded performance for the given problem type to determine the optimal settings.

For small problems, using multiple threads is not be worthwhile and may even be counter productive.

11.1 Presolve

Before an optimizer actually performs the optimization the problem is preprocessed using the so-called presolve. The purpose of the presolve is to

1. remove redundant constraints,
2. eliminate fixed variables,
3. remove linear dependencies,
4. substitute out (implied) free variables, and

5. reduce the size of the optimization problem in general.

After the presolved problem has been optimized the solution is automatically postsolved so that the returned solution is valid for the original problem. Hence, the presolve is completely transparent. For further details about the presolve phase, please see *[AA95]* and *[AGMX96]*.

It is possible to fine-tune the behavior of the presolve or to turn it off entirely. If presolve consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This is done by setting the parameter *presolveUse* to `off`.

The two most time-consuming steps of the presolve are

- the eliminator, and
- the linear dependency check.

Therefore, in some cases it is worthwhile to disable one or both of these.

Numerical issues in the presolve

During the presolve the problem is reformulated so that it hopefully solves faster. However, in rare cases the presolved problem may be harder to solve then the original problem. The presolve may also be infeasible although the original problem is not.

If it is suspected that presolved problem is much harder to solve than the original then it is suggested to first turn the eliminator off by setting the parameter *presolveEliminatorMaxNumTries* to 0. If that does not help, then trying to turn presolve off may help.

Since all computations are done in finite prescision then the presolve employs some tolerances when concluding a variable is fixed or constraint is redundant. If it happens that **MOSEK** incorrectly concludes a problem is primal or dual infeasible, then it is worthwhile to try to reduce the parameters *presolveTolX* and *presolveTolS*. However, if reducing the parameters actually helps then this should be taken as an indication that the problem is badly formulated.

Eliminator

The purpose of the eliminator is to eliminate free and implied free variables from the problem using substitution. For instance, given the constraints

$$
\begin{array}{rcl}
y & = & \sum_j x_j, \\
y, x & \geq & 0,
\end{array}
$$

y is an implied free variable that can be substituted out of the problem, if deemed worthwhile. If the eliminator consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This can be done by setting the parameter *presolveEliminatorMaxNumTries* to 0. In rare cases the eliminator may cause that the problem becomes much hard to solve.

Linear dependency checker

The purpose of the linear dependency check is to remove linear dependencies among the linear equalities. For instance, the three linear equalities

$$
\begin{array}{rcl}
x_1 + x_2 + x_3 & = & 1, \\
x_1 + 0.5x_2 & = & 0.5, \\
0.5x_2 + x_3 & = & 0.5
\end{array}
$$

contain exactly one linear dependency. This implies that one of the constraints can be dropped without changing the set of feasible solutions. Removing linear dependencies is in general a good idea since it reduces the size of the problem. Moreover, the linear dependencies are likely to introduce numerical problems in the optimization phase.

It is best practise to build models without linear dependencies. If the linear dependencies are removed at the modeling stage, the linear dependency check can safely be disabled by setting the parameter *presolveLindepUse* to `off`.

Dualizer

All linear, conic, and convex optimization problems have an equivalent dual problem associated with them. **MOSEK** has built-in heuristics to determine if it is most efficient to solve the primal or dual problem. The form (primal or dual) solved is displayed in the **MOSEK** log. Should the internal heuristics not choose the most efficient form of the problem it may be worthwhile to set the dualizer manually by setting the parameters:

- *intpntSolveForm*: In case of the interior-point optimizer.
- *simSolveForm*: In case of the simplex optimizer.

Note that currently only linear problems may be dualized.

Scaling

Problems containing data with large and/or small coefficients, say $1.0e + 9$ or $1.0e - 7$, are often hard to solve. Significant digits may be truncated in calculations with finite precision, which can result in the optimizer relying on inaccurate calculations. Since computers work in finite precision, extreme coefficients should be avoided. In general, data around the same *order of magnitude* is preferred, and we will refer to a problem, satisfying this loose property, as being *well-scaled*. If the problem is not well scaled, **MOSEK** will try to scale (multiply) constraints and variables by suitable constants. **MOSEK** solves the scaled problem to improve the numerical properties.

The scaling process is transparent, i.e. the solution to the original problem is reported. It is important to be aware that the optimizer terminates when the termination criterion is met on the scaled problem, therefore significant primal or dual infeasibilities may occur after unscaling for badly scaled problems. The best solution of this issue is to reformulate the problem, making it better scaled.

By default **MOSEK** heuristically chooses a suitable scaling. The scaling for interior-point and simplex optimizers can be controlled with the parameters *intpntScaling* and *simScaling* respectively.

11.2 Linear Optimization

11.2.1 Optimizer Selection

Two different types of optimizers are available for linear problems: The default is an interior-point method, and the alternatives are simplex methods. The optimizer can be selected using the parameter *optimizer*.

11.2.2 The Interior-point Optimizer

The purpose of this section is to provide information about the algorithm employed in **MOSEK** interior-point optimizer.

In order to keep the discussion simple it is assumed that **MOSEK** solves linear optimization problems of standard form

$$\begin{array}{lccl} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \geq 0. & & \end{array} \tag{11.1}$$

This is in fact what happens inside **MOSEK**; for efficiency reasons **MOSEK** converts the problem to standard form before solving, then converts it back to the input form when reporting the solution.

Since it is not known beforehand whether problem (11.1) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason that **MOSEK** solves the so-called homogeneous model

$$
\begin{array}{rcl}
Ax - b\tau & = & 0, \\
A^T y + s - c\tau & = & 0, \\
-c^T x + b^T y - \kappa & = & 0, \\
x, s, \tau, \kappa & \geq & 0,
\end{array}
\tag{11.2}
$$

where y and s correspond to the dual variables in (11.1), and τ and κ are two additional scalar variables. Note that the homogeneous model (11.2) always has solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one.

Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (11.2) satisfies

$$x_j^* s_j^* = 0 \text{ and } \tau^* \kappa^* = 0.$$

Moreover, there is always a solution that has the property

$$\tau^* + \kappa^* > 0.$$

First, assume that $\tau^* > 0$. It follows that

$$
\begin{array}{rcl}
A\frac{x^*}{\tau^*} & = & b, \\
A^T \frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} & = & c, \\
-c^T \frac{x^*}{\tau^*} + b^T \frac{y^*}{\tau^*} & = & 0, \\
x^*, s^*, \tau^*, \kappa^* & \geq & 0.
\end{array}
$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left\{ \frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*} \right\}$$

is a primal-dual optimal solution.

On other hand, if $\kappa^* > 0$ then

$$
\begin{array}{rcl}
Ax^* & = & 0, \\
A^T y^* + s^* & = & 0, \\
-c^T x^* + b^T y^* & = & \kappa^*, \\
x^*, s^*, \tau^*, \kappa^* & \geq & 0.
\end{array}
$$

This implies that at least one of

$$-c^T x^* > 0 \tag{11.3}$$

or

$$b^T y^* > 0 \tag{11.4}$$

is satisfied. If (11.3) is satisfied then x^* is a certificate of dual infeasibility, whereas if (11.4) is satisfied then y^* is a certificate of dual infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm *[And09]*.

Interior-point Termination Criterion

For efficiency reasons it is not practical to solve the homogeneous model exactly. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In every iteration, k, of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to homogeneous model is generated where

$$x^k, s^k, \tau^k, \kappa^k > 0.$$

Whenever the trial solution satisfies the criterion

$$\begin{array}{rcl} \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty & \leq & \epsilon_p (1 + \|b\|_\infty), \\ \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty & \leq & \epsilon_d (1 + \|c\|_\infty), \text{ and} \\ \min\left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left|\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}\right| \right) & \leq & \epsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right), \end{array} \tag{11.5}$$

the interior-point optimizer is terminated and

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is reported as the primal-dual optimal solution. The interpretation of (11.5) is that the optimizer is terminated if

- $\frac{x^k}{\tau^k}$ is approximately primal feasible,
- $\left\{\frac{y^k}{\tau^k}, \frac{s^k}{\tau^k}\right\}$ is approximately dual feasible, and
- the duality gap is almost zero.

On the other hand, if the trial solution satisfies

$$-\epsilon_i c^T x^k > \frac{\|c\|_\infty}{\max(1, \|b\|_\infty)} \left\|Ax^k\right\|_\infty$$

then the problem is declared dual infeasible and x^k is reported as a certificate of dual infeasibility. The motivation for this stopping criterion is as follows: First assume that $\left\|Ax^k\right\|_\infty = 0$; then x^k is an exact certificate of dual infeasibility. Next assume that this is not the case, i.e.

$$\left\|Ax^k\right\|_\infty > 0,$$

and define

$$\bar{x} := \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|Ax^k\|_\infty \|c\|_\infty} x^k.$$

It is easy to verify that

$$\|A\bar{x}\|_\infty = \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|c\|_\infty} \text{ and } -c^T\bar{x} > 1,$$

which shows $\bar{x}$ is an approximate certificate of dual infeasibility where ε_i controls the quality of the approximation. A smaller value means a better approximation.

Finally, if

$$\epsilon_i b^T y^k > \frac{\|b\|_\infty}{\max(1, \|c\|_\infty)} \left\|A^T y^k + s^k\right\|_\infty$$

then y^k is reported as a certificate of primal infeasibility.

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see Table 11.1 for details.

Table 11.1: Parameters employed in termination criterion

ToleranceParameter	name
ε_p	*intpntTolPfeas*
ε_d	*intpntTolDfeas*
ε_g	*intpntTolRelGap*
ε_i	*intpntTolInfeas*

The default values of the termination tolerances are chosen such that for a majority of problems appearing in practice it is not possible to achieve much better accuracy. Therefore, tightening the tolerances usually is not worthwhile. However, an inspection of (11.5) reveals that quality of the solution is dependent on $\|b\|_\infty$ and $\|c\|_\infty$; the smaller the norms are, the better the solution accuracy.

The interior-point method as implemented by **MOSEK** will converge toward optimality and primal and dual feasibility at the same rate *[And09]*. This means that if the optimizer is stopped prematurely then it is very unlikely that either the primal or dual solution is feasible. Another consequence is that in most cases all the tolerances, ε_p, ε_d and ε_g, have to be relaxed together to achieve an effect.

In some cases the interior-point method terminates having found a solution not too far from meeting the optimality condition (11.5). A solution is defined as *near optimal* if scaling ε_p, ε_d and ε_g by any number $\varepsilon_n \in [1.0, +\infty]$ conditions (11.5) are satisfied.

A near optimal solution is therefore of lower quality but still potentially valuable. If for instance the solver stalls, i.e. it can make no more significant progress towards the optimal solution, a near optimal solution could be available and be good enough for the user.

The basis identification discussed in Section *Basis Identification* requires an optimal solution to work well; hence basis identification should be turned off if the termination criterion is relaxed.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

Basis Identification

An interior-point optimizer does not return an optimal basic solution unless the problem has a unique primal and dual optimal solution. Therefore, the interior-point optimizer has an optional post-processing step that computes an optimal basic solution starting from the optimal interior-point solution. More information about the basis identification procedure may be found in *[AY96]*. In the following we provide an overall idea of the procedure.

There are some cases in which a basic solution could be more valuable:

- a basic solution is often more accurate than an interior-point solution,
- a basic solution can be used to warm-start the simplex algorithm in case of reoptimization,
- a basic solution is in general more sparse, i.e. more variables are fixed to zero. This is particularly appealing when solving continuous relaxation of mixed integer problems, as well as in all applications in which sparser solutions are preferred.

To illustrate how the basis identification routine works, we use the following trivial example:

$$
\begin{array}{lccc}
\text{minimize} & x + y & & \\
\text{subject to} & x + y & = & 1, \\
& x, y \geq 0. & &
\end{array}
$$

It is easy to see that all feasible solutions are also optimal. In particular, there are two basic solutions namely

$$
\begin{array}{rcl}
(x_1^*, y_1^*) & = & (1, 0), \\
(x_2^*, y_2^*) & = & (0, 1).
\end{array}
$$

The interior point algorithm will actually converge to the center of the optimal set, i.e. to $(x^*, y^*) = (1/2, 1/2)$ (to see this in **MOSEK** deactivate *Presolve*).

In practice, when the algorithm gets close to the optimal solution, it is possible to construct in polynomial time an initial basis for the simplex algorithm from the current interior point solution. This basis is used to warm-start the simplex algorithm that will provide the optimal basic solution.

In most cases the constructed basis is optimal, or very few iterations are required by the simplex algorithm to make it optimal and hence the final *clean* phase be short. However, in some cases for nasty problems e.g. ill-conditioned problems the additional simplex clean up phase may take of lot a time.

By default **MOSEK** performs a basis identification. However, if a basic solution is not needed, the basis identification procedure can be turned off. The parameters

- *intpntBasis*,
- *biIgnoreMaxIter*, and
- *biIgnoreNumError*

control when basis identification is performed.

The type of simplex algorithm to be used can be tuned by the *biCleanOptimizer* parameter i.e. primal or dual simplex, and the maximum number of iterations can be set by the *biMaxIterations*.

Finally, it should be mentioned that there is no guarantee on which basic solution will be returned.

The Interior-point Log

Below is a typical log output from the interior-point optimizer presented:

```
Optimizer  - threads                : 1
Optimizer  - solved problem         : the dual
Optimizer  - Constraints            : 2
Optimizer  - Cones                  : 0
Optimizer  - Scalar variables       : 6                 conic                  : 0
Optimizer  - Semi-definite variables: 0                 scalarized             : 0
Factor     - setup time             : 0.00              dense det. time        : 0.00
Factor     - ML order time          : 0.00              GP order time          : 0.00
Factor     - nonzeros before factor : 3                 after factor           : 3
Factor     - dense dim.             : 0                 flops                  : 7.00e+001
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU       TIME
0   1.0e+000 8.6e+000 6.1e+000 1.00e+000  0.000000000e+000  -2.208000000e+003 1.0e+000 0.00
1   1.1e+000 2.5e+000 1.6e-001 0.00e+000  -7.901380925e+003 -7.394611417e+003 2.5e+000 0.00
2   1.4e-001 3.4e-001 2.1e-002 8.36e-001  -8.113031650e+003 -8.055866001e+003 3.3e-001 0.00
3   2.4e-002 5.8e-002 3.6e-003 1.27e+000  -7.777530698e+003 -7.766471080e+003 5.7e-002 0.01
4   1.3e-004 3.2e-004 2.0e-005 1.08e+000  -7.668323435e+003 -7.668207177e+003 3.2e-004 0.01
5   1.3e-008 3.2e-008 2.0e-009 1.00e+000  -7.668000027e+003 -7.668000015e+003 3.2e-008 0.01
6   1.3e-012 3.2e-012 2.0e-013 1.00e+000  -7.667999994e+003 -7.667999994e+003 3.2e-012 0.01
```

The first line displays the number of threads used by the optimizer and second line tells that the optimizer chose to solve the dual problem rather than the primal problem. The next line displays the problem dimensions as seen by the optimizer, and the `Factor...` lines show various statistics. This is followed by the iteration log.

Using the same notation as in Section *11.2.2* the columns of the iteration log have the following meaning:

- `ITE`: Iteration index.
- `PFEAS`: $\left\| Ax^k - b\tau^k \right\|_\infty$. The numbers in this column should converge monotonically towards to zero but may stall at low level due to rounding errors.
- `DFEAS`: $\left\| A^T y^k + s^k - c\tau^k \right\|_\infty$. The numbers in this column should converge monotonically toward to zero but may stall at low level due to rounding errors.

- GFEAS: $|-c^T x^k + b^T y^k - \kappa^k|$. The numbers in this column should converge monotonically toward to zero but may stall at low level due to rounding errors.
- PRSTATUS: This number converges to 1 if the problem has an optimal solution whereas it converges to -1 if that is not the case.
- POBJ: $c^T x^k / \tau^k$. An estimate for the primal objective value.
- DOBJ: $b^T y^k / \tau^k$. An estimate for the dual objective value.
- MU: $\frac{(x^k)^T s^k + \tau^k \kappa^k}{n+1}$. The numbers in this column should always converge monotonically to zero.
- TIME: Time spend since the optimization started.

11.2.3 The simplex Based Optimizer

An alternative to the interior-point optimizer is the simplex optimizer.

The simplex optimizer uses a different method that allows exploiting an initial guess for the optimal solution to reduce the solution time. Depending on the problem it may be faster or slower to use an initial guess; see section *11.2.4* for a discussion.

MOSEK provides both a primal and a dual variant of the simplex optimizer — we will return to this later.

Simplex Termination Criterion

The simplex optimizer terminates when it finds an optimal basic solution or an infeasibility certificate. A basic solution is optimal when it is primal and dual feasible; see Section *10.1* and *10.1.1* for a definition of the primal and dual problem. Due to the fact that computations are performed in finite precision **MOSEK** allows violation of primal and dual feasibility within certain tolerances. The user can control the allowed primal and dual tolerances with the parameters *basisTolX* and *basisTolS*.

Starting From an Existing Solution

When using the simplex optimizer it may be possible to reuse an existing solution and thereby reduce the solution time significantly. When a simplex optimizer starts from an existing solution it is said to perform a *warm-start*. If the user is solving a sequence of optimization problems by solving the problem, making modifications, and solving again, **MOSEK** will warm-start automatically.

Setting the parameter *optimizer* to freeSimplex instructs **MOSEK** to select automatically between the primal and the dual simplex optimizers. Hence, **MOSEK** tries to choose the best optimizer for the given problem and the available solution.

By default **MOSEK** uses presolve when performing a warm-start. If the optimizer only needs very few iterations to find the optimal solution it may be better to turn off the presolve.

Numerical Difficulties in the Simplex Optimizers

Though **MOSEK** is designed to minimize numerical instability, completely avoiding it is impossible when working in finite precision. **MOSEK** counts a "numerical unexpected behavior" event inside the optimizer as a *set-back*. The user can define how many set-backs the optimizer accepts; if that number is exceeded, the optimization will be aborted. Set-backs are implemented to avoid long sequences where the optimizer tries to recover from an unstable situation.

Set-backs are, for example, repeated singularities when factorizing the basis matrix, repeated loss of feasibility, degeneracy problems (no progress in objective) and other events indicating numerical difficulties. If the simplex optimizer encounters a lot of set-backs the problem is usually badly scaled; in such a situation try to reformulate into a better scaled problem. Then, if a lot of set-backs still occur, trying one or more of the following suggestions may be worthwhile:

- Raise tolerances for allowed primal or dual feasibility: Hence, increase the value of
  - *basisTolX*, and
  - *basisTolS*.
- Raise or lower pivot tolerance: Change the *simplexAbsTolPiv* parameter.
- Switch optimizer: Try another optimizer.
- Switch off crash: Set both *simPrimalCrash* and *simDualCrash* to 0.
- Experiment with other pricing strategies: Try different values for the parameters
  - *simPrimalSelection* and
  - *simDualSelection*.
- If you are using warm-starts, in rare cases switching off this feature may improve stability. This is controlled by the *simHotstart* parameter.
- Increase maximum set backs allowed controlled by *simMaxNumSetbacks*.
- If the problem repeatedly becomes infeasible try switching off the special degeneracy handling. See the parameter *simDegen* for details.

11.2.4 The Interior-point or the Simplex Optimizer?

Given a linear optimization problem, which optimizer is the best: The primal simplex, the dual simplex or the interior-point optimizer?

It is impossible to provide a general answer to this question. However, the interior-point optimizer behaves more predictably: it tends to use between 20 and 100 iterations, almost independently of problem size, but cannot perform warm-start, while simplex can take advantage of an initial solution, but is less predictable for cold-start. The interior-point optimizer is used by default.

11.2.5 The Primal or the Dual Simplex Variant?

MOSEK provides both a primal and a dual simplex optimizer. Predicting which simplex optimizer is faster is impossible, however, in recent years the dual optimizer has seen several algorithmic and computational improvements, which, in our experience, makes it faster on average than the primal simplex optimizer. Still, it depends much on the problem structure and size.

Setting the *optimizer* parameter to `freeSimplex` instructs **MOSEK** to choose which simplex optimizer to use automatically.

To summarize, if you want to know which optimizer is faster for a given problem type, you should try all the optimizers.

11.3 Conic Optimization

11.3.1 The Interior-point Optimizer

For conic optimization problems only an interior-point type optimizer is available. The interior-point optimizer is an implementation of the so-called homogeneous and self-dual algorithm. For a detailed description of the algorithm, please see *[ART03]*.

Interior-point Termination Criteria

The parameters controlling when the conic interior-point optimizer terminates are shown in Table 11.2.

Table 11.2: Parameters employed in termination criterion.

Parameter name	Purpose
intpntCoTolPfeas	Controls primal feasibility
intpntCoTolDfeas	Controls dual feasibility
intpntCoTolRelGap	Controls relative gap
intpntTolInfeas	Controls when the problem is declared infeasible
intpntCoTolMuRed	Controls when the complementarity is reduced enough

11.4 Using Multiple Threads in an Optimizer

If multiple cores are available then it is possible for **MOSEK** to take advantage of them to speed up the computation. However, please note the speedup achieved is going to be dependent on the problem characteristics e.g. the size of problem. Typically for smallish problems no speedup is obtained by exploiting multiple cores. In fact forcing **MOSEK** to use one core can increase speed because parallel overhead is avoided.

11.4.1 Thread Safety

The **MOSEK** API is thread-safe provided that a task is only modified or accessed from one thread at any given time. Also accessing two or more separate tasks from threads at the same time is safe. Sharing an environment between threads is safe.

11.4.2 Determinism

The optimizers are run-to-run deterministic which means if a problem is solved twice on the same computer using the same parameter setting and exactly the same input then exactly the same results is obtained. One qualification is that no time limits must be imposed because the time taken to perform an operation on a computer is dependent on many factors such as the current workload.

11.4.3 The Parallelized Interior-point Optimizer

By default the interior-point optimizer exploits multiple cores using multithreading. Hence, big tasks such as large dense matrix multiplication may be divided into several independent smaller tasks that can be computed independently. However, there is a computational overhead associated with exploiting multiple threads e.g. cost of the additional coordination etc. Therefore, it may be advantageous to turn off the mutithreading for smallish problem using the parameter *intpntMultiThread*.

Moreover, when the interior-point optimizer is allowed to exploit multiple threads, then the parameter *numThreads* controls the maximum number of threads (and therefore the number of cores) that **MOSEK** will employ.

CHAPTER

TWELVE

THE OPTIMIZER FOR MIXED-INTEGER PROBLEMS

A problem is a mixed-integer optimization problem when one or more of the variables are constrained to be integer valued. **MOSEK** can solve mixed-integer

- linear,
- quadratic and quadratically constrained, and
- conic qudratic

problems.

Readers unfamiliar with integer optimization are recommended to consult some relevant literature, e.g. the book *[Wol98]* by Wolsey.

12.1 Some Concepts and Facts Related to Mixed-integer Optimization

It is important to understand that in a worst-case scenario, the time required to solve integer optimization problems grows exponentially with the size of the problem. For instance, assume that a problem contains n binary variables, then the time required to solve the problem in the worst case may be proportional to 2^n . The value of 2^n is huge even for moderate values of n .

In practice this implies that the focus should be on computing a near optimal solution quickly rather than on locating an optimal solution. Even if the problem is only solved approximately, it is important to know how far the approximate solution is from an optimal one. In order to say something about the quality of an approximate solution the concept of *relaxation* is important.

The mixed-integer optimization problem

$$\begin{array}{rclccl} z^* & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \\ & & & x_j \in \mathbb{Z}, & & \forall j \in \mathcal{J}, \end{array} \tag{12.1}$$

has the continuous relaxation

$$\begin{array}{rclccl} \underline{z} & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \end{array} \tag{12.2}$$

The continuos relaxation is identical to the mixed-integer problem with the restriction that some variables must be integers removed.

There are two important observations about the continuous relaxation. First, the continuous relaxation is usually much faster to optimize than the mixed-integer problem. Secondly if $\hat{x}$ is any feasible solution to (12.1) and

$$\bar{z} := c^T \hat{x}$$

then

$$\underline{z} \leq z^* \leq \bar{z}.$$

This is an important observation since if it is only possible to find a near optimal solution within a reasonable time frame then the quality of the solution can nevertheless be evaluated. The value $\underline{z}$ is a lower bound on the optimal objective value. This implies that the obtained solution is no further away from the optimum than $\bar{z} - \underline{z}$ in terms of the objective value.

Whenever a mixed-integer problem is solved **MOSEK** reports this lower bound so that the quality of the reported solution can be evaluated.

12.2 The Mixed-integer Optimizer

The mixed-integer optimizer can handle problems with linear, quadratic objective and constraints and conic constraints. However, a problem can not contain both quadratic objective or constraints and conic constraints.

The mixed-integer optimizer is specialized for solving linear and conic optimization problems. It can also solve pure quadratic and quadratically constrained problems; these problems are automatically converted to conic problems before being solved.

The mixed-integer optimizer is run-to-run deterministic. This means that if a problem is solved twice on the same computer with identical options then the obtained solution will be bit-for-bit identical for the two runs. However, if a time limit is set then this may not be case since the time taken to solve a problem is not deterministic. The mixed-integer optimizer is parallelized i.e. it can exploit multiple cores during the optimization.

The solution process can be split into these phases:

1. **Presolve:** In this phase the optimizer tries to reduce the size of the problem and improve the formulation using preprocessing techniques. The presolve stage can be turned off using the *presolveUse* parameter
2. **Cut generation:** Valid inequalities (cuts) are added to improve the lower bound
3. **Heuristic:** Using heuristics the optimizer tries to guess a good feasible solution. Heuristics can be controlled by the parameter *mioHeuristicLevel*
4. **Search:** The optimal solution is located by branching on integer variables

12.3 Termination Criterion

In general, it is time consuming to find an exact feasible and optimal solution to an integer optimization problem, though in many practical cases it may be possible to find a sufficiently good solution. Therefore, the mixed-integer optimizer employs a relaxed feasibility and optimality criterion to determine when a satisfactory solution is located.

A candidate solution that is feasible for the continuous relaxation is said to be an integer feasible solution if the criterion

$$\min(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j) \leq \delta_1 \quad \forall j \in \mathcal{J}$$

is satisfied, meaning that x_j is at most δ_1 from the nearest integer.

Whenever the integer optimizer locates an integer feasible solution it will check if the criterion

$$\bar{z} - \underline{z} \leq \max(\delta_2, \delta_3 \max(10^{-10}, |\bar{z}|))$$

is satisfied. If this is the case, the integer optimizer terminates and reports the integer feasible solution as an optimal solution. Please note that $\underline{z}$ is a valid lower bound determined by the integer optimizer during the solution process, i.e.

$$\underline{z} \leq z^*.$$

The lower bound $\underline{z}$ normally increases during the solution process.

12.3.1 Relaxed Termination

If an optimal solution cannot be located within a reasonable time, it may be advantageous to employ a relaxed termination criterion after some time. Whenever the integer optimizer locates an integer feasible solution and has spent at least the number of seconds defined by the *mioDisableTermTime* parameter on solving the problem, it will check whether the criterion

$$\bar{z} - \underline{z} \leq \max(\delta_4, \delta_5 \max(10^{-10}, |\bar{z}|))$$

is satisfied. If it is satisfied, the optimizer will report that the candidate solution is **near optimal** and then terminate. Please note that since this criterion depends on timing, the optimizer will not be run to run deterministic.

12.4 Parameters Affecting the Termination of the Integer Optimizer.

All δ tolerances can be adjusted using suitable parameters — see Table 12.1.

Table 12.1: Tolerances for the mixed-integer optimizer.

Tolerance	Parameter name
δ_1	*mioTolAbsRelaxInt*
δ_2	*mioTolAbsGap*
δ_3	*mioTolRelGap*
δ_4	*mioNearTolAbsGap*
δ_5	*mioNearTolRelGap*

In Table 12.2 some other parameters affecting the integer optimizer termination criterion are shown. Please note that if the effect of a parameter is delayed, the associated termination criterion is applied only after some time, specified by the *mioDisableTermTime* parameter.

Table 12.2: Other parameters affecting the integer optimizer termination criterion.

Parameter name	Delayed	Explanation
mioMaxNumBranches	Yes	Maximum number of branches allowed.
mioMaxNumRelaxs	Yes	Maximum number of relaxations allowed.
mioMaxNumSolutions	Yes	Maximum number of feasible integer solutions allowed.

12.5 How to Speed Up the Solution Process

As mentioned previously, in many cases it is not possible to find an optimal solution to an integer optimization problem in a reasonable amount of time. Some suggestions to reduce the solution time are:

- Relax the termination criterion: In case the run time is not acceptable, the first thing to do is to relax the termination criterion — see Section *12.3* for details.

- Specify a good initial solution: In many cases a good feasible solution is either known or easily computed using problem specific knowledge. If a good feasible solution is known, it is usually worthwhile to use this as a starting point for the integer optimizer.
- Improve the formulation: A mixed-integer optimization problem may be impossible to solve in one form and quite easy in another form. However, it is beyond the scope of this manual to discuss good formulations for mixed-integer problems. For discussions on this topic see for example *[Wol98]*.

12.6 Understanding Solution Quality

To determine the quality of the solution one should check the following:

- The solution status key returned by **MOSEK**
- The *optimality gap*: A measure of how much the located solution can deviate from the optimal solution to the problem
- Feasibility. How much the solution violates the constraints of the problem

The *optimality gap* is a measure for how close the solution is to the optimal solution. The optimality gap is given by

$$\epsilon = |(\text{objective value of feasible solution}) - (\text{objective bound})|.$$

The objective value of the solution is guarantied to be within ϵ of the optimal solution.

The optimality gap can be retrieved through the solution item `mioObjAbsGap`. Often it is more meaningful to look at the optimality gap normalized with the magnitude of the solution. The relative optimality gap is available in `mioObjRelGap`.

CHAPTER

THIRTEEN

FUSION API REFERENCE

13.1 Class list

Most commonly used

- *Constraint*: Abstract base class for Constraint objects.
- *Domain*: Base class for variable and constraint domains.
- *Expr*: Represents a linear expression and provides linear operators.
- *Expression*: Abstract base class for all objects which can be used as linear expressions.
- *Matrix*: Base class for all matrix objects.
- *Model*: The object containing all data related to a single optimization model.
- *NDSparseArray*: Representation of a sparse n-dimensional array
- *Set*: Base class shape specification objects.
- *SymmetricLinearDomain*: Represent a linear domain with symmetry.
- *Var*: Provides basic operations on variable objects.
- *Variable*: Abstract base class for Variable objects.

For advanced users

- *BaseSet*: Base class for 1-dimensional sets.
- *BaseVariable*: Abstract base class for Variable objects with default implementations.
- *BoundInterfaceConstraint*: Interface to either the upper bound or the lower bound of a ranged constraint.
- *BoundInterfaceVariable*: Interface to either the upper bound or the lower bound of a ranged variable.
- *CompoundConstraint*: Stacking of contraints.
- *CompoundVariable*: A stack of several other variables.
- *ConicConstraint*: Represent a conic constraint.
- *ConicVariable*: Represent a conic variable.
- *FlatExpr*: A simple sparse representation of a linear expression.
- *LinPSDDomain*: Represent a linear PSD domain.
- *LinearConstraint*: An object representing a block of linear constraints of the same type.
- *LinearDomain*: Represent a domain defined by linear constraints

- *LinearPSDConstraint*: Represents a semidefinite conic constraint.
- *LinearPSDVariable*: This class represents a positive semidefinite variable.
- *LinearVariable*: An object representing a block of linear variables of the same type.
- *ModelConstraint*: Represent a block of constraints.
- *ModelVariable*: Represent a block of variables.
- *PSDConstraint*: Represents a semidefinite conic constraint.
- *PSDDomain*: Represent the domain od PSD matrices.
- *PSDVariable*: This class represents a positive semidefinite variable.
- *PickVariable*: Represents an set of variable entries
- *ProductSet*: None
- *QConeDomain*: A domain representing the Lorentz cone.
- *RangeDomain*: The range domain represents a ranged subset of the euclidian space.
- *RangedConstraint*: Defines a ranged constraint.
- *RangedVariable*: Defines a ranged variable.
- *SliceConstraint*: An alias for a subset of constraints from a single ModelConstraint.
- *SliceVariable*: An alias for a subset of variables from a single model variable.
- *SymLinearVariable*: An object representing a block of linear variables of the same type.
- *SymRangedVariable*: Defines a symmetric ranged variable.
- *SymmetricExpr*: An object representing a symmetric expression.
- *SymmetricRangeDomain*: Represent a ranged domain with symmetry.
- *SymmetricVariable*: An object representing a symmetric variable.

13.1.1 Class BaseSet

`mosek.fusion.BaseSet`

Base class for 1-dimensional sets.

`Members`

BaseSet.dim – Return the size of the given dimension.

Set.compare – Compare two sets and return true if they have the same shape and size.

Set.getSize – Total number of elements in the set.

Set.getname – Return a string representing the index.

Set.idxtokey – Convert a linear index to a N-dimensional key.

Set.realnd – Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

Set.slice – Create a set object representing a slice of this set.

Set.stride – Return the stride size in the given dimension.

Set.toString – Return a string representation of the set.

`Implements`

Set

`ret = BaseSet.dim(i)`

Return the size of the given dimension.

`Parameters`

- `i` (`int32`) – Dimension index.

`Return`

- `ret` (`int32`) – The size of the requested dimension.

13.1.2 Class BaseVariable

`mosek.fusion.BaseVariable`

An abstract variable object. This is class provides various default implementations of methods in *Variable*.

`Members`

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

`Implemented by`

CompoundVariable, *SliceVariable*, *ModelVariable*, *PickVariable*

`ret = BaseVariable.antidiag()`
`ret = BaseVariable.antidiag(index)`

Return the antidiagonal of a square variable matrix.

`Parameters`

- `index` (`int32`) – Index of the anti-diagonal

`Return`

- `ret` (*Variable*)

`ret = BaseVariable.asExpr()`

Create an expression corresponding to the variable object.

`Return`

- `ret` (*Expression*)

`ret = BaseVariable.diag()`
`ret = BaseVariable.diag(index)`

Return the diagonal of a square variable matrix.

Parameters

- index (int32) – Index of the anti-diagonal

Return

- ret (*Variable*)

ret = BaseVariable.dual()
: Get the dual solution value of the variable.

Return

- ret (double[])

ret = BaseVariable.getModel()
: Return the model to which the variable belongs

Return

- ret (*Model*)

ret = BaseVariable.getShape()
: Return the model to which the variable belongs

Return

- ret (*Set*)

ret = BaseVariable.index(index)
ret = BaseVariable.index(index2)
ret = BaseVariable.index(i0, i1)
ret = BaseVariable.index(i0, i1, i2)
: Return a variable slice of size 1 corresponding to a single element in the variable object..

Parameters

- index (int32)
- index2 (int32[])
- i0 (int32) – Index in the first dimension of the element requested.
- i1 (int32) – Index in the second dimension of the element requested.
- i2 (int32) – Index in the second dimension of the element requested.

Return

- ret (*Variable*)

ret = BaseVariable.level()
: Get the primal solution value of the variable.

Return

- ret (double[])

ret = BaseVariable.makeContinuous()
: Drop integrality constraints on the variable, if any

Return

- ret (void)

ret = BaseVariable.makeInteger()
: Apply integrality constraints on the variable

Return

- ret (void)

ret = BaseVariable.pick(idxs)
ret = BaseVariable.pick(midxs)
ret = BaseVariable.pick(i0, i1)
ret = BaseVariable.pick(i0, i1, i2)
: Create a slice variable by picking a list of indexes from this variable.

Parameters

- `idxs` (`int32[]`) – Indexes of the elements requested.
- `midxs` (`int32[][]`) – Matrix of indexes of the elements requested.
- `i0` (`int32[]`)
- `i1` (`int32[]`) – Index along the first dimension.
- `i2` (`int32[]`) – Index along the second dimension.

Return

- `ret` (*Variable*)

`ret = BaseVariable.setLevel(v)`

Input solution values for this variable

Parameters

- `v` (`double[]`) – An array of values to be assigned to the variable.

Return

- `ret` (`void`)

`ret = BaseVariable.shape()`

Return the shape of the variable.

Return

- `ret` (*Set*)

`ret = BaseVariable.size()`

Get the number of elements in the variable.

Return

- `ret` (`int64`)

`ret = BaseVariable.slice(first, last)`
`ret = BaseVariable.slice(first2, last2)`

Create a slice variable by picking a range of indexes for each variable dimension

Parameters

- `first` (`int32`) – The index of the first element(s) of the slice.
- `last` (`int32`) – The index of the first element after the end of the slice.
- `first2` (`int32[]`) – The index of the first element(s) of the slice.
- `last2` (`int32[]`)

Return

- `ret` (*Variable*)

`ret = BaseVariable.toString()`

Create a string-representation of the variable.

Return

- `ret` (`string`)

`ret = BaseVariable.transpose()`

Transpose a vector or matrix variable

Return

- `ret` (*Variable*)

13.1.3 Class BoundInterfaceConstraint

`mosek.fusion.BoundInterfaceConstraint`

Interface to either the upper bound or the lower bound of a ranged constraint.

This class is never explicitly instantiated; is is created by a *RangedConstraint* to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The constraint

$$b_l \leq a^T x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the *RangedConstraint* object, but can be accessed through a *BoundInterfaceConstraint*.

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.toString – Create a human readable representation of the constraint.

SliceConstraint.size – Get the total number of elements in the constraint.

SliceConstraint.slice

Implements

SliceConstraint

13.1.4 Class BoundInterfaceVariable

`mosek.fusion.BoundInterfaceVariable`

Interface to either the upper bound or the lower bound of a ranged variable.

This class is never explicitly instantiated; is is created by a *RangedVariable* to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The variable

$$b_l \leq x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the *RangedVariable* object, but can be accessed through a *BoundInterfaceVariable*.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

SliceVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

`Implements`

SliceVariable

13.1.5 Class CompoundConstraint

`mosek.fusion.CompoundConstraint`

Stacking of constraints.

A *CompoundConstraint* represents a stack or other variable objects and can be used as a 1-dimensional variable. The class is never explicitly instantiated, but is created using *Constraint.stack*.

As this class is derived from *Variable*, it may be used as a normal variable when creating expressions.

`Members`

CompoundConstraint.slice – Unimplemented method!.

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

Constraint.toString – Create a human readable representation of the constraint.

`Implements`

Constraint

`ret = CompoundConstraint.slice(first, last)`
`ret = CompoundConstraint.slice(firsta, lasta)`

Unimplemented method!.

`Parameters`

- `first` (`int32`) – Index of the first element in the slice.
- `last` (`int32`) – Index if the last element in the slice.
- `firsta` (`int32[]`) – Array of start elements in the slice.
- `lasta` (`int32[]`) – Array of end element in the slice.

`Return`

- `ret` (*Constraint*)

13.1.6 Class CompoundVariable

`mosek.fusion.CompoundVariable`

A stack of several other variables.

A compound variable represents a stack og other variable objects and can be used as a 1-dimensional variable. The class is never explicitly instantiated, but is created using *Var.stack*.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

CompoundVariable.asExpr – Create an expression corresponding to the variable object.

CompoundVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

Implements

BaseVariable

`ret = CompoundVariable.asExpr()`

Create an expression corresponding to the variable object.

Return

- `ret` (*Expression*)

`ret = CompoundVariable.slice(first, last)`
`ret = CompoundVariable.slice(first2, last2)`

Create a slice variable by picking a range of indexes for each variable dimension

Parameters

- `first` (`int32`) – The index of the first element(s) of the slice.
- `last` (`int32`) – The index of the first element after the end of the slice.
- `first2` (`int32[]`) – The index of the first element(s) of the slice.
- `last2` (`int32[]`)

Return

- `ret` (*Variable*)

13.1.7 Class ConicConstraint

`mosek.fusion.ConicConstraint`

This class represents a conic constraint of the form

$$Ax - b \in \mathcal{K}$$

where $\mathcal{K}$ is either a quadratic cone or a rotated quadratic cone. Then class is never explicitly instantiated, but is created using *Model.constraint* by specifying a conic domain.

Note that a conic constraint in *Fusion* is always *dense* in the sense that all member constraints are created in the underlying optimization problem immediately.

Members

ConicConstraint.toString – Create a human readable representation of the constraint.

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

ModelConstraint.slice

Implements

ModelConstraint

`ret = ConicConstraint.toString()`

Create a human readable representation of the constraint.

Return

- `ret` (string)

13.1.8 Class ConicVariable

`mosek.fusion.ConicVariable`

This class represents a conic variable of the form

$$Ax - b \in \mathcal{K}$$

where $\mathcal{K}$ is either a quadratic cone or a rotated quadratic cone. Then class is never explicitly instantiated, but is created using *Model.variable* by specifying a conic domain.

Note that a conic variable in *Fusion* is always *dense* in the sense that all member variables are created in the underlying optimization problem immediately.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ConicVariable.toString – Create a string-representation of the variable.

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

`Implements`

ModelVariable

`ret = ConicVariable.toString()`

Create a string-representation of the variable.

`Return`

- `ret` (`string`)

13.1.9 Class Constraint

`mosek.fusion.Constraint`

An abstract constraint object. This is the base class for all constraint types in Fusion.

The *Constraint* object can be an interface to the normal model constraint, e.g. *LinearConstraint* and *ConicConstraint*, to slices of other constraints or to concatenations of other constraints.

Primal and dual solution values can be accessed through the *Constraint* object.

`Members`

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

Constraint.slice

Constraint.toString – Create a human readable representation of the constraint.

`Static Members` *Constraint.stack*

`Implemented by`

CompoundConstraint, *ModelConstraint*, *SliceConstraint*

`ret = Constraint.add(expr)`
`ret = Constraint.add(v)`

ret2 = `Constraint.add(cs)`
ret2 = `Constraint.add(c)`
Add an expression to the constraint expression.

Parameters

- expr (*Expression*)
- v (*Variable*)
- cs (double[])
- c (double)

Return

- ret (*Constraint*) – The constraint itself.
- ret2 (*Constraint*)

ret = `Constraint.dual()`
ret2 = `Constraint.dual(firstidx, lastidx)`
ret3 = `Constraint.dual(firstidx2, lastidx2)`

Parameters

- **firstidx** (int32) – Index of the first element in the range.
- **lastidx** (int32) – Index of the last element (inclusive) in the range.
- **firstidx2** (int32[]) – Array of indexes of the first element in each dimension.
- **lastidx2** (int32[]) – Array of indexes of the last element (inclusive) in each dimension.

Return

- **ret** (double[]) – An array of values corresponding to the dual solution values of the constraint.
- **ret2** (double[]) – An array of solution values.
- **ret3** (double[]) – An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

ret = `Constraint.get_model()`
Get the original model object.

Return

- ret (*Model*) – The model to which the constraint belongs.

ret = `Constraint.get_nd()`
Get the number of dimensions of the constraint.

Return

- ret (int32) – The number of dimensions in the constraint.

ret = `Constraint.index(idx)`
ret = `Constraint.index(idx2)`
Get a single element from a one-dimensional constraint.

Parameters

- idx (int32) – The element index.
- idx2 (int32[]) – Array of integers entry in each dimension.

Return

- ret (*Constraint*) – A new slice containing a single element.

ret = `Constraint.level()`
Get the primal solution value of the variable.

Return

- ret (double[]) – An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

`ret = Constraint.shape()`

Return

- `ret` (*Set*)

`ret = Constraint.size()`

Get the total number of elements in the constraint.

Return

- `ret` (`int64`) – The total numbe of elements in the constraint.

`ret = Constraint.slice(first, last)`
`ret = Constraint.slice(first2, last2)`

Parameters

- `first` (`int32`) – Index of the first element in the slice.
- `last` (`int32`) – Index if the last element in the slice.
- `first2` (`int32[]`) – Array of start elements in the slice.
- `last2` (`int32[]`) – Array of end element in the slice.

Return

- `ret` (*Constraint*) – A new constraint object representing a slice of this object.

`ret = Constraint.stack(v1, v2)`
`ret = Constraint.stack(v1, v2, v3)`
`ret = Constraint.stack(clist)`

Parameters

- `v1` (*Constraint*) – The first constraint in the stack.
- `v2` (*Constraint*) – The second constraint in the stack.
- `v3` (*Constraint*) – The second constraint in the stack.
- `clist` (*Constraint*) – The constraints in the stack.

Return

- `ret` (*Constraint*) – An object representing the concatenation of the constraints.

`ret = Constraint.toString()`

Create a human readable representation of the constraint.

Return

- `ret` (`string`) – A string with the constraint representation.

13.1.10 Class Domain

`mosek.fusion.Domain`

The *Domain* class defines a set of static method for creating various variable and constraint domains. A *Domain* object specifies a subset of $\mathbb{R}^n$, which can be used to define the feasible domain of variables and expressions.

For further details on the use of these, see

- *Model.variable*
- *Model.constraint*

Static Members

Domain.axis – Set the dimension along which the cones are created

Domain.binary – Creates a domain of binary variables.

Domain.equalsTo – Defines the domain consisting of a fixed point.

Domain.greaterThan – Defines the domain consisting of the half-open space bounded below by a value in each dimension.

Domain.inPSDCone – Defines the domain of Positive Semidefinite matrices.

Domain.inQCone – Defines the domain of quadratic cones

Domain.inRange – Creates a domain representing a fixed range in any number of dimensions

Domain.inRotatedQCone – Defines the domain of quadratic cones

Domain.integral – Creates a domain of integral variables.

Domain.isLinPSD – Creates a domain of Positive Semidefinite matrices.

Domain.isTrilPSD – Creates a domain of Positive Semidefinite matrices.

Domain.lessThan – Defines the domain consisting of the half-open space bounded above by a value in each dimension.

Domain.sparse – Ask to use a sparse representation

Domain.symmetric – Impose symmetry on a given linear domain

Domain.unbounded – Creates a domain in which variables are unbounded.

`ret = Domain.axis(c, a)`

Set the dimension along which the cones are created

Parameters

- `c` (*QConeDomain*)
- `a` (`int32`)

Return

- `ret` (*QConeDomain*)

```
ret = Domain.binary(n)
ret = Domain.binary(m, n)
ret = Domain.binary(dims)
ret = Domain.binary()
```

Create a domain composed by n binary variables.

Parameters

- `n` (`int32`) – First dimension size.
- `m` (`int32`) – Second dimension size.
- `dims` (`int32[]`) – A list of dimension sizes.

Return

- `ret` (*RangeDomain*)

```
ret = Domain.equalsTo(b)
ret = Domain.equalsTo(b, n)
ret = Domain.equalsTo(b, m, n)
ret = Domain.equalsTo(b, dims)
ret = Domain.equalsTo(a1)
ret = Domain.equalsTo(a2)
ret = Domain.equalsTo(a1, dims)
ret = Domain.equalsTo(mx)
```

Defines the domain consisting of a fixed point.

Parameters

- `b` (`double`) – A single value. This is scalable: For, say, a $M \times N$ variable, it means that each element in the variable is fixed to b.
- `n` (`int32`) – First dimension size.
- `m` (`int32`) – Second dimension size.

- **a1** (`double[]`) – A one-dimensional array of bounds. The size and shape must match the variable or constraint with which it is used.
- **dims** (`int32[]`) – A list of dimension sizes.
- **a2** (`double[][]`) – A two-dimensional array of bounds. The size and shape must match the variable or constraint with which it is used.
- **mx** (*Matrix*) – A matrix of bound values. The shape must match the variable or constraint with which it is used.

Return

- **ret** (*LinearDomain*)

```
ret = Domain.greaterThan(b)
ret = Domain.greaterThan(b, n)
ret = Domain.greaterThan(b, m, n)
ret = Domain.greaterThan(b, dims)
ret = Domain.greaterThan(a1)
ret = Domain.greaterThan(a2)
ret = Domain.greaterThan(a1, dims)
ret = Domain.greaterThan(mx)
```

Defines the domain consisting of the half-open space bounded below by a value in each dimension.

Parameters

- **b** (`double`) – A single value. This is scalable: For, say, a $M \times N$ variable, it means that each element in the variable is less than to b.
- **n** (`int32`) – First dimension size.
- **m** (`int32`) – Second dimension size.
- **a1** (`double[]`)
- **dims** (`int32[]`) – A list of dimension sizes.
- **a2** (`double[][]`)
- **mx** (*Matrix*) – A matrix of bound values. The shape must match the variable or constraint with which it is used.

Return

- **ret** (*LinearDomain*)

```
ret = Domain.inPSDCone()
ret = Domain.inPSDCone(n)
ret = Domain.inPSDCone(n, m)
```

Creates an object representing m (by default 1) cone(s) of the form:

$$\left\{ X \in \mathbb{R}^{n\times n} | \frac{1}{2}(X + X^T) \in \mathcal{S}_+^n \right\}$$

The shape of the result is $n \times n$ if m was not given, and $n \times n \times m$ if it is..

Parameters

- **n** (`int32`) – Dimension of the cone.
- **m** (`int32`) – Number of cones. By default this is 1.

Return

- **ret** (*PSDDomain*)

```
ret = Domain.inQCone()
ret = Domain.inQCone(n)
ret = Domain.inQCone(m, n)
```

`ret = Domain.inQCone(dims)`

Defines the domain of quadratic cones:

$$\left\{x \in \mathbb{R}^n | x_1^2 \geq \sum_{i=2}^{n} x_i^2,\ x_1 \geq 0\right\}$$

If m is given, it produces a domain definind the set if m cones of this type.

Parameters

- `n` (`int32`) – The size of each cone; at least 2.
- `m` (`int32`) – The number of cones; at least 1.
- `dims` (`int32[]`)

Return

- `ret` (*QConeDomain*)

`ret = Domain.inRange(lb, ub)`
`ret = Domain.inRange(lb, uba)`
`ret = Domain.inRange(lba, ub)`
`ret = Domain.inRange(lba, uba)`
`ret = Domain.inRange(lb, ubm)`
`ret = Domain.inRange(lbm, ub)`
`ret = Domain.inRange(lbm, ubm)`

Create a domain defining a fixed lower and upper bound for each scalar element.

Note that of both upper and lower bounds are defined by a scalar, the resulting domain will scale to any size.

Parameters

- `lb` (`double`) – The lower end of the range as a common scalar value.
- `ub` (`double`) – The upper end of the range as a common scalar value.
- `lba` (`double[]`) – The lower end of the range as an array.
- `uba` (`double[]`) – The upper end of the range as an array.
- `lbm` (*Matrix*) – The lower end of the range as a *Matrix* object.
- `ubm` (*Matrix*) – The upper end of the range as a *Matrix* object.

Return

- `ret` (*RangeDomain*)

`ret = Domain.inRotatedQCone()`
`ret = Domain.inRotatedQCone(n)`
`ret = Domain.inRotatedQCone(m, n)`
`ret = Domain.inRotatedQCone(dims)`

Defines the domain of rotated quadratic cones:

$$\left\{x \in \mathbb{R}^n | x_1 x_2 \geq \sum_{i=3}^{n} x_i^2,\ x_1, x_2 \geq 0\right\}$$

If m is given, it produces a domain definind the set if m cones of this type.

Parameters

- `n` (`int32`) – The size of each cone; at least 3.
- `m` (`int32`) – The number of cones; at least 1.
- `dims` (`int32[]`)

Return

- `ret` (*QConeDomain*)

`ret = Domain.integral(c)`

ret2 = Domain.integral(ld)
ret3 = Domain.integral(rd)

Modify a given domain restricting its elements to be integral.

Parameters

- c (*QConeDomain*) – A conic quadratic domain.
- ld (*LinearDomain*) – A linear domain.
- rd (*RangeDomain*) – A ranged domain.

Return

- ret (*QConeDomain*)
- ret2 (*LinearDomain*)
- ret3 (*RangeDomain*)

ret = Domain.isLinPSD()
ret = Domain.isLinPSD(n)
ret = Domain.isLinPSD(n, m)

Creates an object representing the product of m cones of the form:

$$\left\{X \in \mathbb{R}^{n\times n} | \mathrm{tril}(X) \in \mathcal{S}_+^n\right\}$$

i.e. the lower triangular part of X define the symmetric matrix that is semidefinite.

The shape of the result is $n \times n \times m$.

Parameters

- n (int32) – Dimension of the cone.
- m (int32) – Number of cones. By default this is 1.

Return

- ret (*LinPSDDomain*)

ret = Domain.isTrilPSD()
ret = Domain.isTrilPSD(n)
ret = Domain.isTrilPSD(n, m)

Creates an object representing the product of m cones of the form:

$$\{X \in \mathbb{R}^{n\times n} | \mathrm{tril}(X) \in \mathcal{S}_+^n\}$$

i.e. the lower triangular part of X define the symmetric matrix that is semidefinite.

The shape of the result is $n \times n \times m$.

Parameters

- n (int32) – Dimension of the cone.
- m (int32) – Number of cones. By default this is 1.

Return

- ret (*PSDDomain*)

ret = Domain.lessThan(b)
ret = Domain.lessThan(b, n)
ret = Domain.lessThan(b, m, n)
ret = Domain.lessThan(b, dims)
ret = Domain.lessThan(a1)
ret = Domain.lessThan(a2)
ret = Domain.lessThan(a1, dims)
ret = Domain.lessThan(mx)

Defines the domain consisting of the half-open space bounded above by a value in each dimension.

Parameters

- `b` (`double`) – A single value. This is scalable: For, say, a $M \times N$ variable, it means that each element in the variable is less than to b.
- `n` (`int32`) – First dimension size.
- `m` (`int32`) – Second dimension size.
- `a1` (`double[]`) – A one-dimensional array of bounds. The size and shape must match the variable or constraint with which it is used.
- `dims` (`int32[]`) – A list of dimension sizes.
- `a2` (`double[][]`) – A two-dimensional array of bounds. The size and shape must match the variable or constraint with which it is used.
- `mx` (*`Matrix`*) – A matrix of bound values. The shape must match the variable or constraint with which it is used.

`Return`

- `ret` (*`LinearDomain`*)

`ret = Domain.sparse(ld)`
`ret2 = Domain.sparse(rd)`

Given a linear domain `d`, this method explicitly sugest to *Fusion* that a sparse representation is helpful.

`Parameters`

- `ld` (*`LinearDomain`*) – The putative linear sparse domain
- `rd` (*`RangeDomain`*) – The putative ranged sparse domain

`Return`

- `ret` (*`LinearDomain`*)
- `ret2` (*`RangeDomain`*)

`ret = Domain.symmetric(ld)`
`ret2 = Domain.symmetric(rd)`

Given a linear domain `d`, this method returns a domain such that

$$\{x \in D \subseteq \mathbb{R}^{N \times M} : x_{ij} = x_j i, fori = 1, \ldots, N \quad j = 1, \ldots, M.\}$$

`Parameters`

- `ld` (*`LinearDomain`*) – The linear domain to be modified
- `rd` (*`RangeDomain`*) – The ranged domain to be modified

`Return`

- `ret` (*`SymmetricLinearDomain`*)
- `ret2` (*`SymmetricRangeDomain`*)

`ret = Domain.unbounded()`
`ret = Domain.unbounded(n)`
`ret = Domain.unbounded(m, n)`
`ret = Domain.unbounded(dims)`

Creates a domain in which variables are unbounded.

`Parameters`

- `n` (`int32`) – First dimension size.
- `m` (`int32`) – Second dimension size.
- `dims` (`int32[]`) – A list of dimension sizes.

`Return`

- `ret` (*`LinearDomain`*)

13.1.11 Class Expr

mosek.fusion.Expr

It represents an expression of the form $Ax + b$, where A is a matrix on sparse form, x is a variable vector and b is a vector of scalars.

Additionally, the class defines a set of static method for constructing various expressions.

Members

Expr.eval – Evaluate the expression info to a simple array-based form

Expr.getModel – Return the model to which the expression belongs

Expr.getShape – Return the shape of the expression

Expr.index – Return a specific term of the expression

Expr.numNonzeros – Return the number of non zero elements in the expression.

Expr.pick – Create an expression vector by picking elements from this expression.

Expr.shape – Returns the shape of the expression.

Expr.size – Return the expression size

Expr.slice – Return a slice of the expression

Expr.toString – Create a human readable representation of the expression.

Expr.transpose – Transpose the expression

Static Members

Expr.add – Construct an expression as the sum items.

Expr.constTerm – Create an expression consisting of a constant vector of values.

Expr.dot – Return an object representing the dot-product of two values.

Expr.flatten – Rehshape the expression into a vector

Expr.hstack – Stack a list of expressions horizontally (i.e. along the second dimension).

Expr.mul – Multiply two items.

Expr.mulDiag – Compute the diagonal of the product of two matrixes and return it as a vector.

Expr.mulElm – Element-wise multiplication of two items. The two operands must have the same shape.

Expr.neg – Change the sign of an expression

Expr.ones – Create a vector of ones as an expression.

Expr.outer – Return an object representing the outer-product of two vectors.

Expr.repeat – Repeart an expression a number of times in the given dimension.

Expr.reshape – Reshape the expression into a different shape with the same number of elements.

Expr.stack – Stack a list of expressions in an arbitrary dimension.

Expr.sub – Construct an expression as the difference of two items.

Expr.sum – Sum the elements of an expression

Expr.vstack – Stack a list of expressions vertically (i.e. along the first dimension).

Expr.zeros – Create a vector of zeros as an expression.

```
ret = Expr.add(e1, e2)
ret = Expr.add(e1, v2)
ret = Expr.add(v1, e2)
ret = Expr.add(e1, a1)
```

```
ret = Expr.add(e1, a2)
ret = Expr.add(a1, e2)
ret = Expr.add(a2, e2)
ret = Expr.add(e1, c)
ret = Expr.add(c, e2)
ret = Expr.add(e1, m)
ret = Expr.add(m, e2)
ret = Expr.add(e1, n)
ret = Expr.add(n, e2)
ret = Expr.add(v1, v2)
ret = Expr.add(v1, a1)
ret = Expr.add(v1, a2)
ret = Expr.add(a1, v2)
ret = Expr.add(a2, v2)
ret = Expr.add(v1, c)
ret = Expr.add(c, v2)
ret = Expr.add(v1, m)
ret = Expr.add(m, v2)
ret = Expr.add(v1, n)
ret = Expr.add(n, v2)
ret = Expr.add(vs)
ret = Expr.add(exps)
```

Following combinations of operands are allowed:

A	B
Variable	Variable
Expression	Expression
double	
double[]	
double[,]	
Matrix	
NDSparseArray	

i.e. both `add(A,B)` and `add(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

Parameters

- `e1` (*Expression*) – An expression.
- `e2` (*Expression*) – An expression.
- `a1` (`double[]`) – A one-dimensional array of constants.
- `a2` (`double[][]`) – A two-dimensional array of constants.
- `c` (`double`) – A constant.
- `m` (*Matrix*) – A Matrix object.
- `n` (*NDSparseArray*) – An NDSparseArray object.
- `v1` (*Variable*) – An variable.
- `v2` (*Variable*) – An variable.
- `vs` (*Variable*) – A list of Variables. All variables in the array must have the same shape and size. The list must contain at least one element.
- `exps` (*Expression*) – A list of expressions. All expressions in the array must have the same size. The list must contain at least one element.

Return

- `ret` (*Expression*)

ret = Expr.constTerm(vals1)
ret2 = Expr.constTerm(vals2)
ret = Expr.constTerm(size, val)
ret2 = Expr.constTerm(shp, val2)
ret2 = Expr.constTerm(val2)
ret2 = Expr.constTerm(m)
ret2 = Expr.constTerm(nda)

Create an expression consisting of a constant vector of values.

Parameters

- vals1 (double[]) – Values to put in vector
- vals2 (double[][])
- size (int32) – Length of the vector
- val (double) – Value to put in vector
- shp (*Set*) – Defines the shape of the expression
- val2 (double) – A scalar value to put in vector or matrix expression
- m (*Matrix*) – A Matrix of values to put in the expression
- nda (*NDSparseArray*) – An n-dimensional sparse array of values to put in the expression

Return

- ret (*Expression*) – An expression representing a vector.
- ret2 (*Expression*)

ret = Expr.dot(v, a1)
ret = Expr.dot(v, a2)
ret = Expr.dot(v, m)
ret = Expr.dot(v, spm)
ret = Expr.dot(expr, spm)
ret = Expr.dot(expr, a1)
ret = Expr.dot(expr, a2)
ret = Expr.dot(expr, m)
ret = Expr.dot(a1, expr)
ret = Expr.dot(a1, v)
ret = Expr.dot(a2, expr)
ret = Expr.dot(a2, v)
ret = Expr.dot(spm, expr)
ret = Expr.dot(spm, v)
ret = Expr.dot(m, v)
ret = Expr.dot(m, expr)

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

Parameters

- v (*Variable*) – A variable object.
- a1 (double[]) – A one-dimensional coefficeint array.
- m (*Matrix*) – A matrix object.
- spm (*NDSparseArray*) – A multidimensional sparse array object.
- a2 (double[][]) – A two-dimensional coefficeint array.
- expr (*Expression*) – An expression object.

Return

- ret (*Expression*)

ret = Expr.eval()

Evaluate the expression info to a simple array-based form

Return

- ret (*FlatExpr*)

ret = Expr.flatten(e)

Rehshape the expression into a vector

Parameters

- e (*Expression*)

Return

- ret (*Expression*)

ret = Expr.getModel()

Return the model to which the expression belongs

Return

- ret (*Model*)

ret = Expr.getShape()

Return the shape of the expression

Return

- ret (*Set*)

ret = Expr.hstack(exprs)
ret = Expr.hstack(e1, e2)
ret = Expr.hstack(e1, a2)
ret = Expr.hstack(e1, v2)
ret = Expr.hstack(a1, v2)
ret = Expr.hstack(a1, e2)
ret = Expr.hstack(v1, a2)
ret = Expr.hstack(v1, v2)
ret = Expr.hstack(v1, e2)
ret = Expr.hstack(a1, a2, v3)
ret = Expr.hstack(a1, a2, e3)
ret = Expr.hstack(a1, v2, a3)
ret = Expr.hstack(a1, v2, v3)
ret = Expr.hstack(a1, v2, e3)
ret = Expr.hstack(a1, e2, a3)
ret = Expr.hstack(a1, e2, v3)
ret = Expr.hstack(a1, e2, e3)
ret = Expr.hstack(v1, a2, a3)
ret = Expr.hstack(v1, a2, v3)
ret = Expr.hstack(v1, a2, e3)
ret = Expr.hstack(v1, v2, a3)
ret = Expr.hstack(v1, v2, v3)
ret = Expr.hstack(v1, v2, e3)
ret = Expr.hstack(v1, e2, a3)
ret = Expr.hstack(v1, e2, v3)
ret = Expr.hstack(v1, e2, e3)
ret = Expr.hstack(e1, a2, a3)
ret = Expr.hstack(e1, a2, v3)
ret = Expr.hstack(e1, a2, e3)
ret = Expr.hstack(e1, v2, a3)
ret = Expr.hstack(e1, v2, v3)
ret = Expr.hstack(e1, v2, e3)
ret = Expr.hstack(e1, e2, a3)
ret = Expr.hstack(e1, e2, v3)
ret = Expr.hstack(e12, e22, e32)

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

Parameters

- `exprs` (*Expression*) – A list of expressions.
- `e1` (*Expression*) – An expression, a scalar constant or a variable.
- `e2` (*Expression*) – An expression, a scalar constant or a variable.
- `v3` (*Variable*) – A variable.
- `a1` (`double`) – A scalar constant.
- `a2` (`double`) – a scalar constant.
- `e3` (*Expression*) – An expression, a scalar constant or a variable.
- `v1` (*Variable*) – A variable.
- `v2` (*Variable*) – A variable.
- `a3` (`double`) – a scalar constant.
- `e12` (*Expression*) – An expression object.
- `e22` (*Expression*) – An expression object.
- `e32` (*Expression*) – An expression object.

Return

- `ret` (*Expression*)

`ret = Expr.index(first)`
`ret = Expr.index(firsta)`

Given an expression object e or a variable object v it returns a new expression `-e` and `-v` respectively.

Parameters

- `first` (`int32`) – The index of the terms.
- `firsta` (`int32[]`) – The indexs of the terms.

Return

- `ret` (*Expression*)

`ret = Expr.mul(mx, v)`
`ret2 = Expr.mul(v, mx)`
`ret = Expr.mul(v, vals)`
`ret = Expr.mul(vals, v)`
`ret = Expr.mul(val, v)`
`ret = Expr.mul(v, val)`
`ret = Expr.mul(vals2, v)`
`ret = Expr.mul(v, vals2)`

ret = Expr.mul(expr, val)
ret = Expr.mul(val, expr)
ret3 = Expr.mul(vals, expr)
ret = Expr.mul(expr, vals)
ret = Expr.mul(expr, mx)
ret = Expr.mul(mx, expr)

Following combinations of operands are allowed:

A	B
double	Variable
double[]	Expression
double[,]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

Parameters

- mx (*Matrix*) – A matrix.
- v (*Variable*) – A variable object that may be a scalar or a matrix.
- val (double) – A scalar value.
- vals2 (double[][])
- vals (double[]) – A vector of scalars.
- expr (*Expression*) – An expression object. The shape must match the right-hand side.

Return

- ret (*Expression*)
- ret2 (*Expression*) – A new expression object representing the product of the two operands.
- ret3 (*Expression*) – A new expression.

ret = Expr.mulDiag(a, expr)
ret = Expr.mulDiag(expr, a)
ret = Expr.mulDiag(a, v)
ret = Expr.mulDiag(v, a)
ret2 = Expr.mulDiag(mx, expr2)
ret2 = Expr.mulDiag(expr2, mx)
ret2 = Expr.mulDiag(mx2, v)
ret2 = Expr.mulDiag(v2, mx)

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i\cdot}^t b_{\cdot i}$.

Parameters

- a (double[][])
- expr (*Expression*) – An expression object.
- mx (*Matrix*) – An $m \times n$ matrix object.
- expr2 (*Expression*) – An $n \times p$ expression object.
- mx2 (*Matrix*) – An matrix object.
- v (*Variable*) – A variable object.
- v2 (*Variable*) – An $n \times p$ variable object.

Return

- ret (*Expression*)
- ret2 (*Expression*) – A new *Expr* object.

ret = Expr.mulElm(v, a1)
ret = Expr.mulElm(v, a2)
ret = Expr.mulElm(v, spm)
ret = Expr.mulElm(v, m)
ret = Expr.mulElm(expr, spm)
ret = Expr.mulElm(expr, a1)
ret = Expr.mulElm(expr, a2)
ret = Expr.mulElm(expr, m)
ret = Expr.mulElm(a1, expr)
ret = Expr.mulElm(a1, v)
ret = Expr.mulElm(a2, expr)
ret = Expr.mulElm(a2, v)
ret = Expr.mulElm(spm, expr)
ret = Expr.mulElm(spm, v)
ret = Expr.mulElm(m, v)
ret = Expr.mulElm(m, expr)

Element-wise multiplication of two items. The two operands must have the same shape.

Parameters

- v (*Variable*) – A variable object.
- a1 (double[]) – A one-dimensional coefficeint array.
- spm (*NDSparseArray*) – A multidimensional sparse array object.
- m (*Matrix*) – A matrix object.
- a2 (double[][]) – A two-dimensional coefficeint array.
- expr (*Expression*) – An expression object.

Return

- ret (*Expression*)

ret = Expr.neg(e)
ret = Expr.neg(v)

Given an expression object *e* or a variable object *v* it returns a new expression -e and -v respectively.

Parameters

- e (*Expression*) – An expression object.
- v (*Variable*) – A variable object.

Return

- ret (*Expression*)

ret = Expr.numNonzeros()

Return the number of non zero elements in the expression.

Return

- ret (int64) – The number of non zero elements.

ret = Expr.ones(num)

Create a vector of ones as an expression.

Parameters

- num (int32) – The size of the expression.

Return

- ret (*Expression*) – An expression representing a vector of ones.

ret = Expr.outer(v, a)
ret = Expr.outer(a, v)
ret = Expr.outer(e, a)
ret = Expr.outer(a, e)

Return an object representing the outer product of two vectors.

Parameters

- v (*Variable*) – A vector or matrix variable
- a (double[]) – A vector of constants
- e (*Expression*) – A vector expression

Return

- ret (*Expression*)

`ret = Expr.pick(indexes)`
`ret = Expr.pick(indexrows)`
Create an expression vector by picking elements from this expression.

Parameters

- indexes (int32[]) – A list of integers specifying which indexes to take from an one-dimensional Expression.
- indexrows (int32[][]) – A $n \times m$ array of integers where each row specifies an m-dimensional index to pick.

Return

- ret (*Expression*)

`ret = Expr.repeat(e, n, d)`
Repeart an expression a number of times in the given dimension.

Parameters

- e (*Expression*) – The expression to repeat.
- n (int32) – Number of time to repeat. Must be strictly positive.
- d (int32) – The dimension in which to repeat. Must define a valid dimension index.

Return

- ret (*Expression*)

`ret = Expr.reshape(e, shp)`
`ret = Expr.reshape(e, size)`
`ret = Expr.reshape(e, dimi, dimj)`
Reshape the expression into a different shape with the same number of elements.

Parameters

- e (*Expression*) – The expression to reshape.
- shp (*Set*) – The new shape of the expression; this must have the same total size as the old shape.
- size (int32) – Reshape into a one-dimensional expression of this size.
- dimi (int32) – The first dimension size.
- dimj (int32) – The second dimension size.

Return

- ret (*Expression*)

`ret = Expr.shape()`
Returns the shape of the expression.

Return

- ret (*Set*)

`ret = Expr.size()`
Return the expression size

Return

- ret (int64) – The expression size.

`ret = Expr.slice(first, last)`

ret = `Expr.slice(firsta, lasta)`

Return a slice of the expression

Parameters

- `first` (`int32`) – The index from which the slice begins.
- `last` (`int32`) – The index after the last elements of the slice.
- `firsta` (`int32[]`) – The indexs from which the slice begins.
- `lasta` (`int32[]`) – The indexs after the last elements of the slice.

Return

- `ret` (*Expression*)

```
ret = Expr.stack(dim, exprs)
ret = Expr.stack(dim, e1, e2)
ret = Expr.stack(dim, e1, a2)
ret = Expr.stack(dim, e1, v2)
ret = Expr.stack(dim, a1, v2)
ret = Expr.stack(dim, a1, e2)
ret = Expr.stack(dim, v1, a2)
ret = Expr.stack(dim, v1, v2)
ret = Expr.stack(dim, v1, e2)
ret = Expr.stack(dim, a1, a2, v1)
ret = Expr.stack(dim, a1, a2, e1)
ret = Expr.stack(dim, a1, v2, a3)
ret = Expr.stack(dim, a1, v2, v3)
ret = Expr.stack(dim, a1, v2, e3)
ret = Expr.stack(dim, a1, e2, a3)
ret = Expr.stack(dim, a1, e2, v3)
ret = Expr.stack(dim, a1, e2, e3)
ret = Expr.stack(dim, v1, a2, a3)
ret = Expr.stack(dim, v1, a2, v3)
ret = Expr.stack(dim, v1, a2, e3)
ret = Expr.stack(dim, v1, v2, a3)
ret = Expr.stack(dim, v1, v2, v3)
ret = Expr.stack(dim, v1, v2, e3)
ret = Expr.stack(dim, v1, e2, a3)
ret = Expr.stack(dim, v1, e2, v3)
ret = Expr.stack(dim, v1, e2, e3)
ret = Expr.stack(dim, e1, a2, a3)
ret = Expr.stack(dim, e1, a2, v3)
ret = Expr.stack(dim, e1, a2, e3)
ret = Expr.stack(dim, e1, v2, a3)
ret = Expr.stack(dim, e1, v2, v3)
ret = Expr.stack(dim, e1, v2, e3)
ret = Expr.stack(dim, e1, e2, a3)
ret = Expr.stack(dim, e1, e2, v3)
ret = Expr.stack(dim, e1, e2, e3)
ret = Expr.stack(exprs2)
```

All expressions must have the same shape, except for dimension `dim`. If expressions are

The arguments may be any combination of expressions, scalar constants and variables.

Parameters

- `dim` (`int32`)
- `exprs` (*Expression*) – A list of expressions.
- `e1` (*Expression*) – An expression, a scalar constant or a variable.
- `e2` (*Expression*) – An expression, a scalar constant or a variable.

- a3 (double) – a scalar constant.
- v3 (*Variable*) – A variable.
- e3 (*Expression*) – An expression, a scalar constant or a variable.
- v2 (*Variable*) – A variable.
- a1 (double) – A scalar constant.
- a2 (double) – a scalar constant.
- v1 (*Variable*) – A variable.
- exprs2 (*Expression*) – A list of expressions.

Return

- ret (*Expression*)

```
ret = Expr.sub(e1, e2)
ret = Expr.sub(e1, v2)
ret = Expr.sub(v1, e2)
ret = Expr.sub(e1, a1)
ret = Expr.sub(e1, a2)
ret = Expr.sub(a1, e2)
ret = Expr.sub(a2, e2)
ret = Expr.sub(e1, c)
ret = Expr.sub(c, e2)
ret = Expr.sub(e1, m)
ret = Expr.sub(m, e2)
ret = Expr.sub(e1, n)
ret = Expr.sub(n, e2)
ret = Expr.sub(v1, v2)
ret = Expr.sub(v1, a1)
ret = Expr.sub(v1, a2)
ret = Expr.sub(a1, v2)
ret = Expr.sub(a2, v2)
ret = Expr.sub(v1, c)
ret = Expr.sub(c, v2)
ret = Expr.sub(v1, m)
ret = Expr.sub(m, v2)
ret = Expr.sub(v1, n)
ret = Expr.sub(n, v2)
```

Following combinations of operands are allowed:

A	B
Variable	Variable
Expression	Expression
double	
double[]	
double[,]	
Matrix	
NDSparseArray	

i.e. both sub(A,B) and sub(B,A) are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

Parameters

- e1 (*Expression*) – An expression.
- e2 (*Expression*) – An expression.
- a1 (double[]) – An array of constants.

- a2 (double[][]) – An array of constants.
- c (double)
- m (*Matrix*)
- n (*NDSparseArray*)
- v1 (*Variable*) – An variable.
- v2 (*Variable*) – An variable.

Return

- ret (*Expression*)

ret = Expr.sum(expr)
ret = Expr.sum(v)
ret = Expr.sum(v, d)
ret = Expr.sum(v, dfirst, dlast)
ret = Expr.sum(expr, d)
ret = Expr.sum(expr, dfirst, dlast)

Sum the elements of an expression. Without extra arguments, all elements are summed into an expression of size 1.

With arguments dfirst, dlast or d, elements are summed in a specific dimension or a range of dimensions, resulting in an expression of reduced dimension.

Note that the result of summing over a dimension of size 0 is 0.0. This means that for an expression of shape (2,0,2), summing over the second dimension yields an expression of shape (2,2) of zeros.

Parameters

- expr (*Expression*) – An expression object.
- d (int32) – The dimension to sum.
- v (*Variable*) – An variable.
- dfirst (int32) – The first dimension to sum.
- dlast (int32) – The last-plus-one dimension to sum.

Return

- ret (*Expression*)

ret = Expr.toString()

Create a human readable representation of the expression.

Return

- ret (string) – A string with the representation expression.

ret = Expr.transpose()

Transpose the expression

Return

- ret (*Expression*)

ret = Expr.vstack(exprs)
ret = Expr.vstack(e1, e2)
ret = Expr.vstack(e1, v2)
ret = Expr.vstack(e1, a2)
ret = Expr.vstack(v1, e2)
ret = Expr.vstack(v1, v2)
ret = Expr.vstack(v1, a2)
ret = Expr.vstack(a1, e2)
ret = Expr.vstack(a1, v2)
ret = Expr.vstack(e1, e2, e3)
ret = Expr.vstack(e1, e2, v3)
ret = Expr.vstack(e1, e2, a3)

```
ret = Expr.vstack(e1, v2, e3)
ret = Expr.vstack(e1, v2, v3)
ret = Expr.vstack(e1, v2, a3)
ret = Expr.vstack(e1, a2, e3)
ret = Expr.vstack(e1, a2, v3)
ret = Expr.vstack(e1, a2, a3)
ret = Expr.vstack(v1, e2, e3)
ret = Expr.vstack(v1, e2, v3)
ret = Expr.vstack(v1, e2, a3)
ret = Expr.vstack(v1, v2, e3)
ret = Expr.vstack(v1, v2, v3)
ret = Expr.vstack(v1, v2, a3)
ret = Expr.vstack(v1, a2, e3)
ret = Expr.vstack(v1, a2, v3)
ret = Expr.vstack(v1, a2, a3)
ret = Expr.vstack(a1, e2, e3)
ret = Expr.vstack(a1, e2, v3)
ret = Expr.vstack(a1, e2, a3)
ret = Expr.vstack(a1, v2, e3)
ret = Expr.vstack(a1, v2, v3)
ret = Expr.vstack(a1, v2, a3)
ret = Expr.vstack(a1, a2, e3)
ret = Expr.vstack(a1, a2, v3)
ret = Expr.vstack(a1, a2, a3)
```

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

Parameters

- `exprs` (*Expression*) – A list of expressions.
- `e1` (*Expression*) – An expression, a scalar constant or a variable.
- `e2` (*Expression*) – An expression, a scalar constant or a variable.
- `e3` (*Expression*) – An expression, a scalar constant or a variable.
- `v1` (*Variable*) – A variable.
- `v2` (*Variable*) – A variable.
- `v3` (*Variable*) – A variable.
- `a1` (double) – A scalar constant.
- `a2` (double) – a scalar constant.
- `a3` (double) – a scalar constant.

Return

- `ret` (*Expression*)

ret = Expr.zeros(num)
Create a vector of zeros as an expression.

Parameters

- num (int32) – The size of the expression.

Return

- ret (*Expression*) – An expression representing a vector of zeros.

13.1.12 Class Expression

mosek.fusion.Expression
Abstract base class for all objects which can be used as linear expressions of the form $Ax + b$.

The main use of this class is to store the result of expressions created by the static methods provided by *Expr*.

Members

Expression.eval – Evaluate the expression into simple sparse form.

Expression.getModel – Return the Model object.

Expression.getShape – Initialize the expression as belonging to a given model.

Expression.index – Get a single element of the expression

Expression.pick – Get a list of elements of the expression

Expression.shape – Returns the shape of the expression.

Expression.slice

Expression.toString – Return a string representation of the expression object.

Expression.transpose – Transpose the expression

Implemented by
Expr

ret = Expression.eval()
Evaluate the expression into simple sparse form.

Return

- ret (*FlatExpr*) – The evaluated expression.

ret = Expression.getModel()
Return the Model object.

Return

- ret (*Model*) – The Model object.

ret = Expression.getShape()
Initialize the expression as belonging to a given model.

Return

- ret (*Set*)

ret = Expression.index(i)
ret2 = Expression.index(indexes)
Get a single element of the expression

Parameters

- i (int32) – Index of the element to pick
- indexes (int32[]) – List of indexes of the element to pick

Return

- ret (*Expression*) – A new expression object.

- `ret2` (*Expression*)

`ret = Expression.pick(indexes)`
`ret = Expression.pick(indexrows)`

Get a list of elements of the expression

`Parameters`

- `indexes` (`int32[]`) – Indexes of the elements to pick
- `indexrows` (`int32[][]`) – Indexes of the elements to pick. Each row defines a separate index.

`Return`

- `ret` (*Expression*) – A one dimensional expression object.

`ret = Expression.shape()`

Returns the shape of the expression.

`Return`

- `ret` (*Set*) – A shape object

`ret = Expression.slice(firsta, lasta)`
`ret2 = Expression.slice(first, last)`

`Parameters`

- `firsta` (`int32[]`) – Start of the slice in each dimension
- `lasta` (`int32[]`) – Env of the slice in each dimension
- `first` (`int32`) – Start of the slice
- `last` (`int32`) – Env of the slice

`Return`

- `ret` (*Expression*) – A one dimensional expression object.
- `ret2` (*Expression*) – A new expression object.

`ret = Expression.toString()`

Return a string representation of the expression object.

`Return`

- `ret` (`string`) – A string representation of the object.

`ret = Expression.transpose()`

Transpose the expression

`Return`

- `ret` (*Expression*) – A new expression object.

13.1.13 Class FlatExpr

`mosek.fusion.FlatExpr`

Defines a simple structure containing a sparse representation of a linear expression; basically the result of evaluating an *Expression* object.

`Members`

FlatExpr.size – Get the number of non-zero elements in the expression.

FlatExpr.toString – Create a human readable representation of the expression.

`ret = FlatExpr.size()`

Get the number of non-zero elements in the expression.

`Return`

- `ret` (`int32`) – The number of non-zero elements in the expression.

`ret = FlatExpr.toString()`

Create a human readable representation of the expression.

Return

- ret (string) – A string with the representation expression.

13.1.14 Class LinPSDDomain

mosek.fusion.LinPSDDomain

Represent a linear PSD domain.

13.1.15 Class LinearConstraint

mosek.fusion.LinearConstraint

A linear constraint defines a block of constraints with the same linear domain. The domain is either a product of product of one-dimensional half-spaces (linear inequalities), a fixed value vector (equalities) or the whole space (free constraints).

The *type* of a linear variable is immutable; it is either free, an inequality or an equality, but the linear expression and the right-hand side can be modified.

The class is not meant to be instantiated directly, but must be created by calling the *Model.variable* method.

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

ModelConstraint.slice

ModelConstraint.toString – Create a human readable representation of the constraint.

Implements

ModelConstraint

13.1.16 Class LinearDomain

mosek.fusion.LinearDomain

Represent a domain defined by linear constraints

Members

LinearDomain.integral – Creates a domain of integral variables.

LinearDomain.sparse – Creates a domain exploiting sparsity.

LinearDomain.symmetric – Creates a symmetric domain

ret = LinearDomain.integral()

Creates a domain of integral variables.

Return

- ret (*LinearDomain*)

ret = LinearDomain.sparse()
    Creates a domain exploiting sparsity.

Return

- ret (*LinearDomain*)

ret = LinearDomain.symmetric()
    Creates a symmetric domain

Return

- ret (*SymmetricLinearDomain*)

13.1.17 Class LinearPSDConstraint

mosek.fusion.LinearPSDConstraint
    This class represents a semidefinite conic constraint of the form

$$Ax - b \succeq 0$$

i.e. $Ax - b$ must be positive semidefinite

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

LinearPSDConstraint.toString – Create a human readable representation of the constraint.

ModelConstraint.slice

Implements
    ModelConstraint

ret = LinearPSDConstraint.toString()
    Create a human readable representation of the constraint.

Return

- ret (string)

13.1.18 Class LinearPSDVariable

mosek.fusion.LinearPSDVariable
    This class represents a positive semidefinite variable.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

LinearPSDVariable.toString – Create a string-representation of the variable.

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

Implements

ModelVariable

ret = LinearPSDVariable.toString()

Create a string-representation of the variable.

Return

- ret (string)

13.1.19 Class LinearVariable

mosek.fusion.LinearVariable

A linear variable defines a block of variables with the same linear domain. The domain is either a product of product of one-dimensional half-spaces (linear inequalities), a fixed value vector (equalities) or the whole space (free variables).

The *type* of a linear variable is immutable; it is either free, an inequality or an equality.

The class is not meant to be instantiated directly, but must be created by calling the *Model.variable* method.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

`Implements`

ModelVariable

13.1.20 Class Matrix

`mosek.fusion.Matrix`

Base class for all matrix objects.

`Members`

Matrix.get – Get non-zero at position (i,j).

Matrix.getDataAsArray – Return the data a dense array of values.

Matrix.getDataAsTriplets – Return the matrix data in triplet format.

Matrix.isSparse – Returns true if the matrix is sparse.

Matrix.numColumns – Returns the number columns in the matrix.

Matrix.numNonzeros – Returns the number of non-zeros in the matrix.

Matrix.numRows – Returns the number rows in the matrix.

Matrix.toString – Get a string representation of the matrix.

Matrix.transpose – Transpose the matrix.

`Static Members`

Matrix.antidiag – Create a sparse square matrix with a given vector as anti-diagonal

Matrix.dense – Create a sparse square matrix with a given vector as anti-diagonal

Matrix.diag – Create a sparse square matrix with a given vector as diagonal

Matrix.eye – Create the identity matrix.

Matrix.ones – Create a matrix filled with all ones.

Matrix.sparse – Create a sparse square matrix with a given vector as anti-diagonal

`ret = Matrix.antidiag(d)`
`ret = Matrix.antidiag(d, k)`
`ret = Matrix.antidiag(n, val)`
`ret = Matrix.antidiag(n, val, k)`

Create a sparse square matrix with a given vector as anti-diagonal

`Parameters`

- `d` (`double[]`) – The diagonal vector
- `k` (`int32`) – The diagonal index. $k = 0$ is the default and means the main diagonal. $k > 0$ means diagonals above the main, and $k < 0$ means the diagonals below the main.
- `n` (`int32`) – The size of each side in the matrix.
- `val` (`double`) – Use this value for all diagonal elements.

`Return`

- ret (*Matrix*)

ret = Matrix.dense(data)
ret = Matrix.dense(dimi, dimj, data2)
ret = Matrix.dense(dimi, dimj, value)
ret = Matrix.dense(other)

Create a sparse square matrix with a given vector as anti-diagonal

Parameters

- data (double[][]) – A one- or two-dimensional array of matrix coefficients.
- dimi (int32) – Number of rows in matrix.
- dimj (int32) – Number of columns in matrix.
- data2 (double[]) – A one- or two-dimensional array of matrix coefficients.
- value (double) – Use this value for all elements.
- other (*Matrix*) – Create a dense matrix from another matrix.

Return

- ret (*Matrix*)

ret = Matrix.diag(d)
ret = Matrix.diag(d, k)
ret = Matrix.diag(n, val)
ret = Matrix.diag(n, val, k)
ret2 = Matrix.diag(md)
ret = Matrix.diag(num, mv)

Create a sparse square matrix with a given vector as diagonal

Parameters

- d (double[]) – The diagonal vector
- k (int32) – The diagonal index. $k = 0$ is the default and means the main diagonal. $k > 0$ means diagonals above the main, and $k < 0$ means the diagonals below the main.
- n (int32) – The size of each side in the matrix.
- val (double) – Use this value for all diagonal elements.
- md (*Matrix*) – A list of square matrixes that are used to create a block-diagonal square matrix.
- num (int32) – Number of times to repeat the mv matrix.
- mv (*Matrix*)

Return

- ret (*Matrix*)
- ret2 (*Matrix*) – A sparse block diagonal matrix.

ret = Matrix.eye(n)

Create the identity matrix.

Parameters

- n (int32)

Return

- ret (*Matrix*) – The identity matrix of size n.

ret = Matrix.get(i, j)

Get non-zero at position (i,j).

Parameters

- i (int32)
- j (int32)

Return

- `ret` (`double`)

`ret` = `Matrix.getDataAsArray()`
Return the data a dense array of values.

`Return`

- `ret` (`double[]`)

`ret` = `Matrix.getDataAsTriplets(subi, subj, val)`
Return the matrix data in triplet format.

`Parameters`

- `subi` (`int32[]`) – Row subscripts are returned in this array.
- `subj` (`int32[]`) – Column subscripts are returned in this array.
- `val` (`double[]`) – Coefficient values are returned in this arary.

`Return`

- `ret` (`void`)

`ret` = `Matrix.isSparse()`
Returns true if the matrix is sparse.

`Return`

- `ret` (`bool`)

`ret` = `Matrix.numColumns()`
Returns the number columns in the matrix.

`Return`

- `ret` (`int32`) – The number of columns.

`ret` = `Matrix.numNonzeros()`
Returns the number of non-zeros in the matrix.

`Return`

- `ret` (`int64`) – The number of non-zeros.

`ret` = `Matrix.numRows()`
Returns the number rows in the matrix.

`Return`

- `ret` (`int32`) – The number of rows.

`ret` = `Matrix.ones(n, m)`
Create a matrix filled with all ones.

`Parameters`

- `n` (`int32`)
- `m` (`int32`)

`Return`

- `ret` (*`Matrix`*) – An $n \times m$ matrix fileld by ones.

`ret` = `Matrix.sparse(nrow, ncol, subi, subj, val)`
`ret` = `Matrix.sparse(subi, subj, val)`
`ret` = `Matrix.sparse(subi, subj, val2)`
`ret` = `Matrix.sparse(nrow, ncol, subi, subj, val2)`
`ret` = `Matrix.sparse(nrow, ncol)`
`ret` = `Matrix.sparse(data)`
`ret` = `Matrix.sparse(blocks)`
`ret` = `Matrix.sparse(mx)`
Create a sparse square matrix with a given vector as anti-diagonal

`Parameters`

- `nrow` (`int32`)

- `ncol` (`int32`)
- `subi` (`int32[]`) – Row subscripts of non-zero elements.
- `subj` (`int32[]`) – Column subscripts of non-zero elements.
- `val` (`double[]`) – Coefficients of non-zero elements.
- `val2` (`double`) – Coefficients of non-zero elements.
- `data` (`double[][]`) – Dense data array.
- `blocks` (*Matrix*) – The matrix data. This is a two-dimensional array of *Matrix* objects or NULL. In `blocks`, all elements in a row must have the same height, and all elements in a column must have the same width.

  Entries that are NULL will be interpreted as a block of zeros whose height and width are deduced from the other elements in the same row and column. Any row that contains only NULL entries will have height 0, and any column that contains only NULL entries will have width 0.
- `mx` (*Matrix*) – A *Matrix* object

Return

- `ret` (*Matrix*)

`ret = Matrix.toString()`

Get a string representation of the matrix.

Return

- `ret` (`string`) – A string representation of the matrix.

`ret = Matrix.transpose()`

Transpose the matrix.

Return

- `ret` (*Matrix*)

13.1.21 Class Model

`mosek.fusion.Model`

The object containing all data related to a single optimization model.

Members

Model.acceptedSolutionStatus – Get or set the accepted solution status.

Model.addConstraint – Add a set of constraints to one that is already in the model.

Model.addVariable – Add a set of variable to one that is already in the model.

Model.breakSolver – Request that the solver terminates as soon as possible.

Model.clone – Clone the model.

Model.constraint – Create a new constraint in the model.

Model.dispose – Destroy the Model object

Model.dualObjValue – Get the dual objective value.

Model.flushSolutions – If any solution values have been inputted, flush those values to the underlying task.

Model.getAcceptedSolutionStatus – Get the the accepted solution status.

Model.getConstraint – Get the constraint corresponding to the given name or index

Model.getDualSolutionStatus

Model.getName – Return the model name, or an empty string if it has not been set.

Model.getPrimalSolutionStatus

Model.getProblemStatus – Return the problem status

Model.getSolverDoubleInfo – Fetch a solution information item from the solver

Model.getSolverIntInfo – Fetch a solution information item from the solver

Model.getSolverLIntInfo – Fetch a solution information item from the solver

Model.getTask – Return the underlying MOSEK task object.

Model.getVariable – Get the variable corresponding to the given name or index

Model.hasConstraint – Return whether the model contains a constraint with a given name.

Model.hasVariable – Return whether the model contains a variable with a given name.

Model.numConstraints – Return the number of constraints

Model.numVariables – Return the number of variables

Model.objective – Replace the objective expression.

Model.primalObjValue – Get the primal objective value.

Model.selectedSolution – Set which solution to take values from.

Model.setCallbackHandler – Attach a callback handler.

Model.setLogHandler – Attach a log handler.

Model.setSolverParam – Set a solver parameter

Model.solve – Attempt to optimize the model.

Model.variable – Create a new variable in the model.

Model.writeTask – Dump the current solver task to a file.

Static Members

Model.putlicensecode – Set the license code in the global environment

Model.putlicensepath – Set the license path in the global environment

Model.putlicensewait – Set the license wait flag in the global environment

Implements
BaseModel

ret = Model.acceptedSolutionStatus(what)

This sets or gets the flag that indicated what solutions are accepted as *expected* when fetching primal and dual solution values.

When fetching a solution value the status of the solution is checked against the flag. If it matches, the solution is returned, otherwise an exception is thrown. The two methods *Model.getPrimalSolutionStatus* and *Model.getDualSolutionStatus* can be used to get the *actual* status of the solutions.

By default the accepted solution status is *NearOptimal*.

Parameters

- what (*AccSolutionStatus*) – The new accepted solution status.

Return

- ret (void)

ret = Model.addConstraint(name, v)

Add a set of constraints to one that is already in the model.

Parameters

- name (string) – The name of the constraint set in the model
- v (*ModelConstraint*) – The new contraint set

Return

- ret (void)

ret = Model.addVariable(name, v)
Add a set of variable to one that is already in the model.

Parameters

- name (string) – The name of the variable set in the model
- v (*ModelVariable*) – The new variable set

Return

- ret (void)

ret = Model.breakSolver()
Request that the solver terminates as soon as possible.

Return

- ret (void)

ret = Model.clone()
Clone the model.

Return

- ret (*Model*)

```
ret = Model.constraint(name, expr, dom)
ret = Model.constraint(expr, dom)
ret = Model.constraint(name, expr, dom2)
ret = Model.constraint(expr, dom2)
ret = Model.constraint(name, shape, expr, dom3)
ret = Model.constraint(shape, expr, dom3)
ret = Model.constraint(name, expr, dom3)
ret = Model.constraint(expr, dom3)
ret = Model.constraint(name, shape, expr, dom4)
ret = Model.constraint(shape, expr, dom4)
ret = Model.constraint(name, expr, dom4)
ret = Model.constraint(expr, dom4)
ret = Model.constraint(name, shape, expr, dom5)
ret = Model.constraint(shape, expr, dom5)
ret = Model.constraint(name, expr, dom5)
ret = Model.constraint(expr, dom5)
ret = Model.constraint(name, v, dom)
ret = Model.constraint(v, dom)
ret = Model.constraint(name, v, dom2)
ret = Model.constraint(v, dom2)
ret = Model.constraint(name, shape, v, dom3)
ret = Model.constraint(shape, v, dom3)
ret = Model.constraint(name, v, dom3)
ret = Model.constraint(v, dom3)
ret = Model.constraint(name, shape, v, dom4)
ret = Model.constraint(shape, v, dom4)
ret = Model.constraint(name, v, dom4)
ret = Model.constraint(v, dom4)
ret = Model.constraint(name, shape, v, dom5)
ret = Model.constraint(shape, v, dom5)
ret = Model.constraint(name, v, dom5)
ret = Model.constraint(v, dom5)
```

Parameters

- name (string) – Name of the constraint. This must be unique among all constraints in the model. The value NULL is allowed instead of a unique name.

- expr (*Expression*) – An expression.
- dom (*PSDDomain*) – Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.
- dom2 (*LinPSDDomain*) – Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.
- dom4 (*RangeDomain*) – Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.
- dom5 (*QConeDomain*) – Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.
- shape (*Set*) – Defines the shape of the constraint. If this is NULL, the shape will be derived from the shape of `expr`.
- v (*Variable*) – A variable used as an expression.
- dom3 (*LinearDomain*) – Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.

Return

- ret (*Constraint*)

ret = `Model.dispose()`

Destroy the Model object

Return

- ret (void)

ret = `Model.dualObjValue()`

Get the dual objective value.

Return

- ret (double)

ret = `Model.flushSolutions()`

If any solution values have been inputted, flush those values to the underlying task.

Return

- ret (void)

ret = `Model.getAcceptedSolutionStatus()`

Get the the accepted solution status.

Return

- ret (*AccSolutionStatus*)

ret = `Model.getConstraint(name)`
ret = `Model.getConstraint(index)`

Get the constraint corresponding to the given name or index

Parameters

- name (string) – The constraint name
- index (int32) – The constraint index

Return

- ret (*Constraint*)

ret = `Model.getDualSolutionStatus(which)`
ret2 = `Model.getDualSolutionStatus()`

Parameters

- which (*SolutionType*) – the type of the solution (see *SolutionType*)

Return

- ret (*SolutionStatus*) – The dual solution *SolutionStatus*

- ret2 (*SolutionStatus*)

ret = Model.getName()
Return the model name, or an empty string if it has not been set.

Return

- ret (string) – The model name.

ret = Model.getPrimalSolutionStatus(which)
ret2 = Model.getPrimalSolutionStatus()

Parameters

- which (*SolutionType*) – the type of the solution (see *SolutionType*)

Return

- ret (*SolutionStatus*) – The primal solution *SolutionStatus*
- ret2 (*SolutionStatus*)

ret = Model.getProblemStatus(which)
Return the problem status

Parameters

- which (*SolutionType*) – the type of the solution (see *SolutionType*)

Return

- ret (*ProblemStatus*) – The problem status *ProblemStatus*

ret = Model.getSolverDoubleInfo(name)
Fetch a solution information item from the solver

Parameters

- name (string) – A string identifying the information to be fetched.

Return

- ret (double)

ret = Model.getSolverIntInfo(name)
Fetch a solution information item from the solver

Parameters

- name (string)

Return

- ret (int32)

ret = Model.getSolverLIntInfo(name)
Fetch a solution information item from the solver

Parameters

- name (string)

Return

- ret (int64)

ret = Model.getTask()
Return the underlying MOSEK task object.

Return

- ret (Task)

ret = Model.getVariable(name)
ret = Model.getVariable(index)
Get the variable corresponding to the given name or index

Parameters

- name (string) – The variable name
- index (int32) – The variable index

Return

- `ret` (*Variable*)

`ret = Model.hasConstraint(name)`
Return whether the model contains a constraint with a given name.

`Parameters`

- `name` (`string`) – The constraint name

`Return`

- `ret` (bool)

`ret = Model.hasVariable(name)`
Return whether the model contains a variable with a given name.

`Parameters`

- `name` (`string`) – The variable name

`Return`

- `ret` (bool)

`ret = Model.numConstraints()`
Return the number of constraints

`Return`

- `ret` (int64)

`ret = Model.numVariables()`
Return the number of variables

`Return`

- `ret` (int64)

`ret = Model.objective(name, sense, expr)`
`ret = Model.objective(name, sense, v)`
`ret = Model.objective(name, sense, c)`
`ret = Model.objective(name, c)`
`ret = Model.objective(sense, expr)`
`ret = Model.objective(sense, v)`
`ret = Model.objective(sense, c)`
`ret = Model.objective(c)`
Replace the objective expression.

`Parameters`

- `name` (`string`) – Name of the obective; this may be any string, and its has no function except when writing the problem to an external file formal.
- `sense` (*ObjectiveSense*) – The objective sense; defines whether the objective must be minimized or maximized.
- `expr` (*Expression*) – The objective expression. This must be an expression containing exactly one row.
- `v` (*Variable*) – The objective variable. This must be a scaler variable.
- `c` (double)

`Return`

- `ret` (void)

`ret = Model.primalObjValue()`
Get the primal objective value.

`Return`

- `ret` (double)

`ret = Model.putlicensecode(code)`
Set the license code in the global environment

`Parameters`

- code (int32[])

Return

- ret (void)

ret = Model.putlicensepath(licfile)
Set the license path in the global environment

Parameters

- licfile (string)

Return

- ret (void)

ret = Model.putlicensewait(wait)
Set the license wait flag in the global environment

Parameters

- wait (bool)

Return

- ret (void)

ret = Model.selectedSolution(soltype)
Set which solution to take values from.

Parameters

- soltype (*SolutionType*)

Return

- ret (void)

ret = Model.setCallbackHandler(h)
Attach a callback handler.

Parameters

- h (System.CallbackHandler) – The callback handler or NULL.

Return

- ret (void)

ret = Model.setLogHandler(h)
Attach a log handler.

Parameters

- h (System.StreamWriter) – The log handler object or NULL.

Return

- ret (void)

ret = Model.setSolverParam(name, strval)
ret = Model.setSolverParam(name, intval)
ret = Model.setSolverParam(name, floatval)
Solver parameter values can be either symbolic values, integers or doubles, depending on the parameter. The value is automatically converted to a suitable type, or, if this fails, an exception will be thrown. For example, if the parameter accepts a double value and is give a string, the string will be parsed to produce a double.

See *13.4.1* for a listing of all parameter settings.

Parameters

- name (string) – Name of the parameter to set
- strval (string) – A string value to assign to the parameter.
- intval (int32) – An integer value to assign to the parameter.
- floatval (double) – A float value to assign to the parameter.

Return

- ret (void)

ret = Model.solve()
Attempt to optimize the model.

Return

- ret (void)

ret = Model.variable(name)
ret = Model.variable(name, size)
ret = Model.variable(name, size2)
ret = Model.variable(name, size, dom)
ret = Model.variable(name, size, dom2)
ret = Model.variable(name, size, dom3)
ret = Model.variable(name, shp, dom)
ret = Model.variable(name, shp, dom2)
ret = Model.variable(name, shp, dom3)
ret = Model.variable(name, size2, dom)
ret = Model.variable(name, size2, dom2)
ret = Model.variable(name, dom)
ret = Model.variable(name, dom2)
ret = Model.variable(name, dom3)
ret = Model.variable()
ret = Model.variable(size)
ret = Model.variable(size2)
ret = Model.variable(size, dom)
ret = Model.variable(size, dom2)
ret = Model.variable(size, dom3)
ret = Model.variable(shp, dom)
ret = Model.variable(shp, dom2)
ret = Model.variable(shp, dom3)
ret = Model.variable(size2, dom)
ret = Model.variable(size2, dom2)
ret = Model.variable(dom)
ret = Model.variable(dom2)
ret = Model.variable(dom3)
ret2 = Model.variable(name, size, dom4)
ret2 = Model.variable(size, dom4)
ret = Model.variable(name, shp, dom5)
ret = Model.variable(name, n, dom5)
ret = Model.variable(name, n, m, dom5)
ret = Model.variable(name, dom5)
ret = Model.variable(n, dom5)
ret = Model.variable(n, m, dom5)
ret = Model.variable(dom5)
ret = Model.variable(name, shp, dom6)
ret = Model.variable(name, n, dom6)
ret = Model.variable(name, n, m, dom6)
ret = Model.variable(name, dom6)
ret = Model.variable(n, dom6)
ret = Model.variable(n, m, dom6)
ret = Model.variable(dom6)

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom)
```

where any or all of name and shp can be left out, and dom can be either a *Domain* or a *RangeDomain*.

Parameters

- name (string) – Name of the variable. This must be unique among all variables in the model. The value NULL is allowed instead of a unique name.
- size (int32) – Size of the variable. The variable becomes a one-dimensional vector of the given size.
- dom (*LinearDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.
- dom4 (*SymmetricLinearDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.
- shp (*Set*) – Defines the shape of the variable.
- size2 (int32[]) – Size of the variable. The variable becomes a one-dimensional vector of the given size.
- dom2 (*RangeDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.
- dom3 (*QConeDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.
- dom5 (*PSDDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.
- n (int32)
- m (int32)
- dom6 (*LinPSDDomain*) – Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. *Domain.equalsTo* (0.0), or the size and shape must be matched by the shape defined by either shape or size.

Return

- ret (*Variable*)
- ret2 (*SymmetricVariable*)

ret = Model.writeTask(filename)

Dump the current solver task to a file.

Parameters

- filename (string) – Name of the file to write.

Return

- ret (void)

13.1.22 Class ModelConstraint

mosek.fusion.ModelConstraint

Base class for all constraints that directly corresponds to a block of constraints in the underlying task, i.e. all objects created from *Model.constraint* .

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

ModelConstraint.slice

ModelConstraint.toString – Create a human readable representation of the constraint.

`Implements`
Constraint

`Implemented by`
ConicConstraint, *RangedConstraint*, *LinearConstraint*, *PSDConstraint*, *LinearPSDConstraint*

`ret = ModelConstraint.slice(first, last)`
`ret = ModelConstraint.slice(first2, last2)`

`Parameters`

- `first` (`int32`) – Index of the first element in the slice.
- `last` (`int32`) – Index if the last element in the slice.
- `first2` (`int32[]`) – Array of start elements in the slice.
- `last2` (`int32[]`) – Array of end element in the slice.

`Return`

- `ret` (*Constraint*)

`ret = ModelConstraint.toString()`
Create a human readable representation of the constraint.

`Return`

- `ret` (`string`)

13.1.23 Class ModelVariable

`mosek.fusion.ModelVariable`
Base class for all variables that directly corresponds to a block of variables in the underlying task, i.e. all objects created from *Model.variable*.

`Members`

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

`Implements`

BaseVariable

`Implemented by`

ConicVariable, *LinearPSDVariable*, *RangedVariable*, *LinearVariable*, *PSDVariable*, *SymLinearVariable*, *SymRangedVariable*

`ret = ModelVariable.slice(first, last)`
`ret = ModelVariable.slice(first2, last2)`

Create a slice variable by picking a range of indexes for each variable dimension

`Parameters`

- `first` (`int32`) – The index of the first element(s) of the slice.
- `last` (`int32`) – The index of the first element after the end of the slice.
- `first2` (`int32[]`) – The index of the first element(s) of the slice.
- `last2` (`int32[]`)

`Return`

- `ret` (*Variable*)

13.1.24 Class NDSparseArray

`mosek.fusion.NDSparseArray`

Representation of a sparse n-dimensional array

`Static Members` *NDSparseArray.make* Create a sparse n-dimensional matrix (tensor)

`ret = NDSparseArray.make(dims, sub, cof)`
`ret = NDSparseArray.make(dims, inst, cof)`
`ret = NDSparseArray.make(m)`

Create a sparse n-dimensional matrix (tensor)

`Parameters`

- `dims` (`int32[]`) – Array dimensions
- `sub` (`int32[][]`) – Array of non-zero n-dimensional subscripts
- `cof` (`double[]`) – Array of coefficients corresponding to subscripts
- `inst` (`int64[]`) – Array of linear indexes of non-zero subscripts
- `m` (*Matrix*)

`Return`

- `ret` (*NDSparseArray*)

13.1.25 Class PSDConstraint

`mosek.fusion.PSDConstraint`

This class represents a semidefinite conic constraint of the form

$$Ax - b \succeq 0$$

i.e. $Ax - b$ must be positive semidefinite

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

ModelConstraint.slice

PSDConstraint.toString – Create a human readable representation of the constraint.

Implements

ModelConstraint

`ret = PSDConstraint.toString()`

Create a human readable representation of the constraint.

Return

- `ret` (string)

13.1.26 Class PSDDomain

`mosek.fusion.PSDDomain`

Represent the domain od PSD matrices.

13.1.27 Class PSDVariable

`mosek.fusion.PSDVariable`

This class represents a positive semidefinite variable.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

PSDVariable.toString – Create a string-representation of the variable.

`Implements`

ModelVariable

`ret = PSDVariable.toString()`

Create a string-representation of the variable.

`Return`

- `ret` (`string`)

13.1.28 Class PickVariable

`mosek.fusion.PickVariable`

Represents an set of variable entries

`Members`

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

PickVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

`Implements`

BaseVariable

`ret = PickVariable.slice(first, last)`
`ret = PickVariable.slice(first2, last2)`

Create a slice variable by picking a range of indexes for each variable dimension

`Parameters`

- `first` (`int32`) – The index of the first element(s) of the slice.
- `last` (`int32`) – The index of the first element after the end of the slice.

- `first2` (`int32[]`) – The index of the first element(s) of the slice.
- `last2` (`int32[]`)

`Return`

- `ret` (*Variable*)

13.1.29 Class ProductSet

`mosek.fusion.ProductSet`

One-dimensional set defined as a range of integers.

`Members` *ProductSet.indexToString*

`ret = ProductSet.indexToString(index)`

`Parameters`

- `index` (int64)

`Return`

- `ret` (string)

13.1.30 Class QConeDomain

`mosek.fusion.QConeDomain`

A domain representing the Lorentz cone.

`Members`

QConeDomain.axis – Set the dimension along which the cones are created.

QConeDomain.getAxis – Get the dimension along which the cones are created.

QConeDomain.integral – Creates a domain of integral variables.

`ret = QConeDomain.axis(a)`

Set the dimension along which the cones are created.

`Parameters`

- `a` (int32)

`Return`

- `ret` (*QConeDomain*)

`ret = QConeDomain.getAxis()`

Get the dimension along which the cones are created.

`Return`

- `ret` (int32)

`ret = QConeDomain.integral()`

Creates a domain of integral variables.

`Return`

- `ret` (*QConeDomain*)

13.1.31 Class RangeDomain

`mosek.fusion.RangeDomain`

The *RangeDomain* object is never instantiated directly: Instead use the relevant methods in *Domain*.

`Members`

RangeDomain.integral – Creates a domain of integral variables.

RangeDomain.sparse – Creates a domain exploiting sparsity.

RangeDomain.symmetric – Creates a symmetric domain.

`Implemented by`

SymmetricRangeDomain

`ret = RangeDomain.integral()`

Creates a domain of integral variables.

`Return`

- `ret` (*RangeDomain*)

`ret = RangeDomain.sparse()`

Creates a domain exploiting sparsity.

`Return`

- `ret` (*RangeDomain*)

`ret = RangeDomain.symmetric()`

Creates a symmetric domain.

`Return`

- `ret` (*SymmetricRangeDomain*) – A new domain

13.1.32 Class RangedConstraint

`mosek.fusion.RangedConstraint`

Defines a ranged constraint.

Since this actually defines one constraint with two inequalities, there will be two dual values (slc and suc) corresponding to the lower and upper bounds. When asked for the dual solution, this constraint will return (y=slc-suc), but in some cases this is not enough (the individual dual constraints may be required for a certificate of infeasibility). The methods *RangedConstraint.lowerBoundCon* and *RangedConstraint.upperBoundCon* returns Variable objects that interface to the lower and upper bounds respectively.

`Members`

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.size – Get the total number of elements in the constraint.

ModelConstraint.slice

ModelConstraint.toString – Create a human readable representation of the constraint.

RangedConstraint.lowerBoundCon – Get a constraint object corresponding to the lower bound of the ranged constraint.

RangedConstraint.upperBoundCon – Get a constraint object corresponding to the upper bound of the ranged constraint.

`Implements`

ModelConstraint

`ret = RangedConstraint.lowerBoundCon()`

Get a constraint object corresponding to the lower bound of the ranged constraint.

`Return`

- `ret` (*Constraint*) – A new constraint object representing the lower bound of the constraint.

`ret = RangedConstraint.upperBoundCon()`
Get a constraint object corresponding to the upper bound of the ranged constraint.

`Return`

- `ret` (*Constraint*) – A new constraint object representing the upper bound of the constraint.

13.1.33 Class RangedVariable

`mosek.fusion.RangedVariable`
Defines a ranged variable.

Since this actually defines one variable with two inequalities, there will be two dual variables (slx and sux) corresponding to the lower and upper bounds. When asked for the dual solution, this variable will return (y=slx-sux), but in some cases this is not enough (the individual dual variables may be required by e.g. a certificate). The methods *RangedVariable.lowerBoundVar* and *RangedVariable.upperBoundVar* returns Variable objects that interface to the lower and upper bounds respectively.

`Members`

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

RangedVariable.lowerBoundVar – Get a variable object corresponding to the lower bound of the ranged variable.

RangedVariable.upperBoundVar – Get a variable object corresponding to the upper bound of the ranged variable.

`Implements`
ModelVariable

`ret = RangedVariable.lowerBoundVar()`
Get a variable object corresponding to the lower bound of the ranged variable.

Return

- ret (*Variable*) – A variable object representing the lower bound of the variable.

ret = RangedVariable.upperBoundVar()
Get a variable object corresponding to the upper bound of the ranged variable.

Return

- ret (*Variable*) – A variable object representing the upper bound of the variable.

13.1.34 Class Set

mosek.fusion.Set
Base class shape specification objects.

Members

Set.compare – Compare two sets and return true if they have the same shape and size.

Set.dim – Return the size of the given dimension.

Set.getSize – Total number of elements in the set.

Set.getname – Return a string representing the index.

Set.idxtokey – Convert a linear index to a N-dimensional key.

Set.realnd – Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

Set.slice – Create a set object representing a slice of this set.

Set.stride – Return the stride size in the given dimension.

Set.toString – Return a string representation of the set.

Static Members

Set.make – Creates a set object

Set.scalar – Create a set of size 1

Implemented by
BaseSet

ret = Set.compare(other)
Compare two sets and return true if they have the same shape and size.

Parameters

- other (*Set*) – The set to compare against.

Return

- ret (bool) – Whether the two set are equal.

ret = Set.dim(i)
Return the size of the given dimension.

Parameters

- i (int32) – Dimension index.

Return

- ret (int32) – The size of the requested dimension.

ret = Set.getSize()
Total number of elements in the set.

Return

- ret (int64) – The number of elements.

ret = Set.getname(key)

ret = Set.getname(keya)

Return a string representing the index.

Parameters

- key (int64) – A linear index.
- keya (int32[]) – An N-dimensional index.

Return

- ret (string) – Get a string representing the item identified by the key.

ret = Set.idxtokey(idx)

Convert a linear index to a N-dimensional key.

Parameters

- idx (int64) – A linear index.

Return

- ret (int32[]) – The N-dimensional key for the linear index.

ret = Set.make(names)
ret = Set.make(sz)
ret = Set.make(s1, s2)
ret = Set.make(s1, s2, s3)
ret = Set.make(sizes)
ret = Set.make(s12, s22)
ret = Set.make(ss)

This static method is a factory for different kind of set objects:

- A (multi-dimensional) set of integers.
- A set whose elements are strings.
- A set obtained as Cartesian product of sets given in a list.

Parameters

- names (string[]) – A list of strings
- sz (int32) – The dimension for a integer set
- s1 (int32) – Size of the first dimension
- s2 (int32) – Size of the second dimension
- s3 (int32) – Size of the third dimension
- sizes (int32[]) – The sizes of dimensions for a integer set
- s12 (*Set*) – Size of the first dimension
- s22 (*Set*) – Size of the second dimension
- ss (*Set*) – A list of sets

Return

- ret (*Set*)

ret = Set.realnd()

Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

Return

- ret (int32) – The number of dimensions.

ret = Set.scalar()

Create a set of size 1

Return

- ret (*Set*) – The new set.

ret = Set.slice(first, last)

ret = Set.slice(firsta, lasta)
Create a set object representing a slice of this set.

Parameters

- first (int32) – First index in the range.
- last (int32) – Last index in the range.
- firsta (int32[]) – First index in each dimension in the range.
- lasta (int32[]) – Last index in each dimension in the range.

Return

- ret (*Set*) – A new *Set* object representing the slice.

ret = Set.stride(i)
Return the stride size in the given dimension.

Parameters

- i (int32) – Dimension index.

Return

- ret (int64) – The stride size in the requested dimension.

ret = Set.toString()
Return a string representation of the set.

Return

- ret (string) – A string representation of the set.

13.1.35 Class SliceConstraint

mosek.fusion.SliceConstraint
An alias for a subset of constraints from a single ModelConstraint.

This class acts as a proxy for accessing a portion of a ModelConstraint. It is possible to access and modify the properties of the original variable using this alias. It does not access the Model directly, only through the original variable.

Members

Constraint.add – Add an expression to the constraint expression.

Constraint.dual

Constraint.get_model – Get the original model object.

Constraint.get_nd – Get the number of dimensions of the constraint.

Constraint.index – Get a single element from a one-dimensional constraint.

Constraint.level – Get the primal solution value of the variable.

Constraint.shape

Constraint.toString – Create a human readable representation of the constraint.

SliceConstraint.size – Get the total number of elements in the constraint.

SliceConstraint.slice

Implements

Constraint

Implemented by

BoundInterfaceConstraint

ret = SliceConstraint.size()
Get the total number of elements in the constraint.

Return

- ret (int64)

ret = SliceConstraint.slice(firstidx, lastidx)
ret = SliceConstraint.slice(firstidx2, lastidx2)
Parameters

- firstidx (int32)
- lastidx (int32)
- firstidx2 (int32[])
- lastidx2 (int32[])

Return

- ret (*Constraint*)

13.1.36 Class SliceVariable

mosek.fusion.SliceVariable
An alias for a subset of variables from a single *ModelVariable*.

This class acts as a proxy for accessing a portion of a *ModelVariable*. It is possible to access and modify the properties of the original variable using this alias, and the object can be used in expressions as any other *Variable* object.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.toString – Create a string-representation of the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

SliceVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

Implements
BaseVariable

Implemented by
BoundInterfaceVariable

ret = SliceVariable.slice(firstidx, lastidx)

ret = SliceVariable.slice(firstidx2, lastidx2)
Create a slice variable by picking a range of indexes for each variable dimension

Parameters

- firstidx (int32)
- lastidx (int32)
- firstidx2 (int32[])
- lastidx2 (int32[])

Return

- ret (*Variable*)

13.1.37 Class SymLinearVariable

mosek.fusion.SymLinearVariable
A linear variable defines a block of variables with the same linear domain. The domain is either a product of product of one-dimensional half-spaces (linear inequalities), a fixed value vector (equalities) or the whole space (free variables).

The *type* of a linear variable is immutable; it is either free, an inequality or an equality.

The class is not meant to be instantiated directly, but must be created by calling the *Model.variable* method.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

SymLinearVariable.toString – Create a string-representation of the variable.

Implements
ModelVariable

ret = SymLinearVariable.toString()
Create a string-representation of the variable.

Return

- ret (string)

13.1.38 Class SymRangedVariable

mosek.fusion.SymRangedVariable

Defines a ranged variable.

Since this actually defines one variable with two inequalities, there will be two dual variables (slx and sux) corresponding to the lower and upper bounds. When asked for the dual solution, this variable will return (y=slx-sux), but in some cases this is not enough (the individual dual variables may be required by e.g. a certificate). The methods *RangedVariable.lowerBoundVar* and *RangedVariable.upperBoundVar* returns Variable objects that interface to the lower and upper bounds respectively.

Members

BaseVariable.antidiag – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.getModel – Return the model to which the variable belongs

BaseVariable.getShape – Return the model to which the variable belongs

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any

BaseVariable.makeInteger – Apply integrality constraints on the variable

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.shape – Return the shape of the variable.

BaseVariable.size – Get the number of elements in the variable.

BaseVariable.transpose – Transpose a vector or matrix variable

ModelVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

SymRangedVariable.toString – Create a string-representation of the variable.

Implements

ModelVariable

ret = SymRangedVariable.toString()

Create a string-representation of the variable.

Return

- ret (string)

13.1.39 Class SymmetricExpr

mosek.fusion.SymmetricExpr

A guaranteed symmetric square matrix expression.

It is defined as

$$\sum_i (M_i x_i) + b,$$

$$where : math : `M_i`isa : msk : func : `SymmetricMatrix`and : math : `x_i`isascalarvariable.$$

Members *SymmetricExpr.toString* Returns a human readable representation of the expression.

ret = SymmetricExpr.toString()
Returns a human readable representation of the expression.

Return

- ret (string) – A string representing the expression.

13.1.40 Class SymmetricLinearDomain

mosek.fusion.SymmetricLinearDomain
Represent a linear domain with symmetry.

Members

SymmetricLinearDomain.integral – Creates a domain of integral variables.

SymmetricLinearDomain.sparse – Creates a domain exploiting sparsity.

ret = SymmetricLinearDomain.integral()
Creates a domain of integral variables.

Return

- ret (*SymmetricLinearDomain*)

ret = SymmetricLinearDomain.sparse()
Creates a domain exploiting sparsity.

Return

- ret (*SymmetricLinearDomain*)

13.1.41 Class SymmetricRangeDomain

mosek.fusion.SymmetricRangeDomain
Represent a ranged domain with symmetry.

Members

RangeDomain.integral – Creates a domain of integral variables.

RangeDomain.sparse – Creates a domain exploiting sparsity.

RangeDomain.symmetric – Creates a symmetric domain.

Implements

RangeDomain

13.1.42 Class SymmetricVariable

mosek.fusion.SymmetricVariable
An object representing a symmetric variable.

Members

Variable.antidiag – Return the antidiagonal of a square variable matrix.

Variable.asExpr – Create an expression corresponding to the variable object.

Variable.diag – Return the diagonal of a square variable matrix.

Variable.dual – Return the dual value of the variable as an array.

Variable.getModel – Return the model to which the variable belongs

Variable.getShape – Return the model to which the variable belongs

Variable.index – Return a variable slice of size one corresponding to a single element in the variable object.

Variable.level – Return the primal value of the variable as an array.

Variable.makeContinuous – Drop integrality constraints on the variable, if any

Variable.makeInteger – Apply integrality constraints on the variable

Variable.pick – Create a slice variable by picking a list of indexes from this variable.

Variable.setLevel – Input solution values for this variable

Variable.shape – Return the shape of the variable.

Variable.size – Get the number of elements in the variable.

Variable.slice – Create a slice variable by picking a range of indexes for each variable dimension.

Variable.toString – Create a string-representation of the variable.

Variable.transpose – Transpose a vector or matrix variable

`Implements`

Variable

`Implemented by`

PSDVariable, *SymLinearVariable*, *SymRangedVariable*

13.1.43 Class Var

`mosek.fusion.Var`

An abstract variable object. This is the base class for all variable types in Fusion, and it contains several static methods for manipulating variable objects.

The *Variable* object can be an interface to the normal model variables, e.g. *LinearVariable* and *ConicVariable*, to slices of other variables or to concatenations of other variables.

Primal and dual solution values can be accessed through the *Variable* object.

`Static Members`

Var.compress – Reshape a variable object by removing all dimensions of size 1.

Var.flatten – Create a one-dimensional logical view of a variable object.

Var.hrepeat – Create a variable by repeating a variable in the second dimension.

Var.hstack – Create a stacked variable in second dimension.

Var.repeat – Create a variable by repeating a variable in a given dimension.

Var.reshape – Create a reshaped version of the given variable.

Var.stack – Create a stacked variable in dimension `dim`.

Var.vrepeat – Create a variable by repeating a variable in a first dimension.

Var.vstack – Create a stacked variable in first dimension.

`ret = Var.compress(v)`

Reshape a variable object by removing all dimensions of size 1.

`Parameters`

- v (*Variable*) – The variable object to reshape.

`Return`

- ret (*Variable*)

ret = Var.flatten(v)
    Create a one-dimensional logical view of a variable object.

Parameters

- v (*Variable*) – The variable to be flattened

Return

- ret (*Variable*) – A one-dimensional Variable object.

ret = Var.hrepeat(v, n)
    Create a variable by repeating a variable in the second dimension.

Parameters

- v (*Variable*) – A variable object.
- n (int32) – Number of times to repeat v.

Return

- ret (*Variable*)

ret = Var.hstack(v)
ret = Var.hstack(v1, v2)
ret = Var.hstack(v1, v2, v3)
    Create a stacked variable in second dimension.

Parameters

- v (*Variable*) – List of variable to stack.
- v1 (*Variable*) – The first variable in the stack.
- v2 (*Variable*) – The second variable in the stack.
- v3 (*Variable*) – The third variable in the stack.

Return

- ret (*Variable*) – An object representing the concatenation of the variables.

ret = Var.repeat(v, dim, n)
ret = Var.repeat(v, n)
    Create a variable by repeating a variable in the given dimension. For an m-dimensional variable with the shape $(d_1, \ldots, d_m)$: If the dimension, dim is negative, a new first dimension is inserted of size n, and the result will have shape $(n, d_1, \ldots, d_m)$. Otherwise the result will have shape $(d_1, \ldots, d_{\texttt{dim}}, \ldots, d_m)$.

    By default it will repeat in the first dimension.

Parameters

- v (*Variable*) – A variable object.
- dim (int32) – Dimension to repeat in. If this is negative, it means that the result adds a new dimension.
- n (int32) – Number of times to repeat v.

Return

- ret (*Variable*)

ret = Var.reshape(v, s)
ret2 = Var.reshape(v2, dims)
ret2 = Var.reshape(v2, d1, d2)
ret2 = Var.reshape(v2, d1)
    Create a reshaped version of the given variable.

Parameters

- v (*Variable*) – The variable to be reshaped
- s (*Set*) – The new shape of the variable

- v2 (*Variable*) – A variable object.
- dims (int32[]) – An array containing the shape of the new variable.
- d1 (int32) – Size of first dimension in the result.
- d2 (int32) – Size of second dimension in the result.

Return

- ret (*Variable*) – A new variable object with the shape defined by s
- ret2 (*Variable*)

ret = Var.stack(v, dim)
ret = Var.stack(v1, v2, dim)
ret = Var.stack(v1, v2, v3, dim)
ret2 = Var.stack(vlist)

Create a stacked variable in dimension dim.

Parameters

- v (*Variable*) – List of variable to stack.
- dim (int32) – Dimension in which to stack.
- v1 (*Variable*) – First variable in the stack.
- v2 (*Variable*) – Second variable in the stack.
- v3 (*Variable*) – Third variable in the stack.
- vlist (*Variable*) – The variables in the stack.

Return

- ret (*Variable*)
- ret2 (*Variable*) – An object representing the concatenation of the variables.

ret = Var.vrepeat(v, n)

Create a variable by repeating a variable in the first dimension.

Parameters

- v (*Variable*) – A variable object.
- n (int32) – Number of times to repeat v.

Return

- ret (*Variable*)

ret = Var.vstack(v)
ret = Var.vstack(v1, v2)
ret = Var.vstack(v1, v2, v3)

Create a stacked variable in first dimension.

Parameters

- v (*Variable*) – List of variable to stack.
- v1 (*Variable*) – First variable in the stack.
- v2 (*Variable*) – Second variable in the stack.
- v3 (*Variable*) – Third variable in the stack.

Return

- ret (*Variable*) – An object representing the concatenation of the variables.

13.1.44 Class Variable

`mosek.fusion.Variable`

An abstract variable object. This is the base class for all variable types in Fusion, and it contains several static methods for manipulating variable objects.

The *Variable* object can be an interface to the normal model variables, e.g. *LinearVariable* and *ConicVariable*, to slices of other variables or to concatenations of other variables.

Primal and dual solution values can be accessed through the *Variable* object.

Members

Variable.antidiag – Return the antidiagonal of a square variable matrix.

Variable.asExpr – Create an expression corresponding to the variable object.

Variable.diag – Return the diagonal of a square variable matrix.

Variable.dual – Return the dual value of the variable as an array.

Variable.getModel – Return the model to which the variable belongs

Variable.getShape – Return the model to which the variable belongs

Variable.index – Return a variable slice of size one corresponding to a single element in the variable object.

Variable.level – Return the primal value of the variable as an array.

Variable.makeContinuous – Drop integrality constraints on the variable, if any

Variable.makeInteger – Apply integrality constraints on the variable

Variable.pick – Create a slice variable by picking a list of indexes from this variable.

Variable.setLevel – Input solution values for this variable

Variable.shape – Return the shape of the variable.

Variable.size – Get the number of elements in the variable.

Variable.slice – Create a slice variable by picking a range of indexes for each variable dimension.

Variable.toString – Create a string-representation of the variable.

Variable.transpose – Transpose a vector or matrix variable

Implemented by

BaseVariable, *SymmetricVariable*

`ret = Variable.antidiag(index)`
`ret = Variable.antidiag()`

Return the antidiagonal of a square variable matrix.

Parameters

- `index` (`int32`) – Defining the index of the anti-diagonal. 0 is the diagonal starting at element (1,n). Positive values are the super-diagonals (diagonals in the upper triangular part, and negative are indexes of the sub-diagonals (in the lower triangular part).

Return

- `ret` (*Variable*)

`ret = Variable.asExpr()`

Create an expression corresponding to the variable object.

Return

- `ret` (*Expression*) – An Expression object representing the V variable.

`ret = Variable.diag(index)`
`ret = Variable.diag()`

Return the diagonal of a square variable matrix.

Parameters

- `index` (`int32`) – Defining the index of the diagonal. 0 is the diagonal starting at element (1,1). Positive values are the super-diagonals (diagonals in the upper triangular part, and negative are indexes of the sub-diagonals (in the lower triangular part).

Return

- `ret` (*Variable*)

`ret = Variable.dual()`

Return the dual value of the variable as an array.

Return

- `ret` (`double[]`) – An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

`ret = Variable.getModel()`

Return the model to which the variable belongs

Return

- `ret` (*Model*)

`ret = Variable.getShape()`

Return the model to which the variable belongs

Return

- `ret` (*Set*)

`ret = Variable.index(i1)`
`ret = Variable.index(i1, i2)`
`ret = Variable.index(i1, i2, i3)`
`ret = Variable.index(idx)`

Return a variable slice of size one corresponding to a single element in the variable object.

Parameters

- `i1` (`int32`) – Index in the first dimension of the element requested.
- `i2` (`int32`) – Index in the second dimension of the element requested.
- `i3` (`int32`) – Index in the third dimension of the element requested.
- `idx` (`int32[]`) – List of indexes of the elements requested.

Return

- `ret` (*Variable*)

`ret = Variable.level()`

Return the primal value of the variable as an array.

Return

- `ret` (`double[]`) – An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

`ret = Variable.makeContinuous()`

Drop integrality constraints on the variable, if any

Return

- `ret` (void)

`ret = Variable.makeInteger()`

Apply integrality constraints on the variable

Return

- `ret` (void)

`ret = Variable.pick(idxs)`
`ret = Variable.pick(midxs)`
`ret = Variable.pick(i1, i2)`

ret = Variable.pick(i1, i2, i3)
Create a slice variable by picking a list of indexes from this variable.

Parameters

- idxs (int32[]) – Indexes of the elements requested.
- midxs (int32[][]) – Matrix of indexes of the elements requested.
- i1 (int32[]) – Index along the first dimension.
- i2 (int32[]) – Index along the second dimension.
- i3 (int32[]) – Index along the third dimension.

Return

- ret (*Variable*)

ret = Variable.setLevel(v)
Input solution values for this variable

Parameters

- v (double[]) – An array of values to be assigned to the variable.

Return

- ret (void)

ret = Variable.shape()
Return the shape of the variable.

Return

- ret (*Set*) – A set representing the shape.

ret = Variable.size()
Get the number of elements in the variable.

Return

- ret (int64)

ret = Variable.slice(first, last)
ret = Variable.slice(firsta, lasta)
Create a slice variable by picking a range of indexes for each variable dimension.

Parameters

- first (int32) – The index of the first element of the slice.
- last (int32) – The index of the first element after the end of the slice.
- firsta (int32[]) – The indexes of the first elements of the slice along each dimension.
- lasta (int32[]) – The indexes of the first elements after the end of the slice along each dimension.

Return

- ret (*Variable*) – A new variable object representing a slice of this object.

ret = Variable.toString()
Create a string-representation of the variable.

Return

- ret (string) – A string representing the variable.

ret = Variable.transpose()
Transpose a vector or matrix variable

Return

- ret (*Variable*) – A new variable object.

13.2 Exceptions

- *DimensionError*: Thrown when a given object has the wrong number of dimensions, or they have not the right size.
- *DomainError*: Invalid domain.
- *ExpressionError*: Tried to construct an expression from invalid.
- *FatalError*: A fatal error has happened.
- *FusionException*: Base class for all normal exceptions in fusion.
- *FusionRuntimeException*: Base class for all run-time exceptions in fusion.
- *IOError*: Error when reading or writing a stream, or opening a file.
- *IndexError*: Index out of bound, or a multi-dimensional index had wrong number of dimensions.
- *LengthError*: None
- *MatrixError*: Thrown if data used in construction of a matrix contained inconsistencies or errors.
- *ModelError*: Thrown when objects from different models were mixed.
- *NameError*: Name clash; tries to add a variable or constraint with a name that already exists.
- *OptimizeError*: An error occurred during optimization.
- *ParameterError*: Tried to use an invalid parameter for a value that was invalid for a specific parameter.
- *RangeError*: Invalid range specified
- *SetDefinitionError*: Invalid data for constructing set.
- *SliceError*: Invalid slice definition, negative slice or slice index out of bounds.
- *SolutionError*: Requested a solution that was undefined or whose status was not acceptable.
- *SparseFormatError*: The given sparsity patters was invalid or specified an index that was out of bounds.
- *UnexpectedError*: An unexpected error has happened. No specific excepion could have been risen.
- *UnimplementedError*: Called a stub. Functionality has not yet been implemented.
- *ValueConversionError*: Error casting or converting a value.

13.2.1 Exception DimensionError

`mosek.fusion.DimensionError`

Thrown when a given object has the wrong number of dimensions, or they have not the right size.

`Members` *FusionRuntimeException.toString* Return the exception message.

`Implements`

FusionRuntimeException

13.2.2 Exception DomainError

`mosek.fusion.DomainError`

Invalid domain.

`Members` *FusionRuntimeException.toString* Return the exception message.

`Implements`

FusionRuntimeException

13.2.3 Exception ExpressionError

`mosek.fusion.ExpressionError`

Tried to construct an expression from invalid.

`Members` *FusionRuntimeException.toString* Return the exception message.

`Implements`

FusionRuntimeException

13.2.4 Exception FatalError

`mosek.fusion.FatalError`

A fatal error has happened.

`Members` `RuntimeException.toString` Return the exception message.

`Implements`

`RuntimeException`

13.2.5 Exception FusionException

`mosek.fusion.FusionException`

Base class for all normal exceptions in fusion.

`Members` *FusionException.toString* Return the exception message.

`Implements`

`Exception`

`Implemented by`

SolutionError

`ret = FusionException.toString()`

Return the exception message.

`Return`

- `ret` (`string`) – The message.

13.2.6 Exception FusionRuntimeException

`mosek.fusion.FusionRuntimeException`

Base class for all run-time exceptions in fusion.

`Members` *FusionRuntimeException.toString* Return the exception message.

`Implements`

`RuntimeException`

`Implemented by`

ValueConversionError, *IndexError*, *ParameterError*, *SetDefinitionError*, *OptimizeError*, *NameError*, *IOError*, *RangeError*, *ModelError*, *DomainError*, *ExpressionError*, *LengthError*, *MatrixError*, *SparseFormatError*, *SliceError*, *DimensionError*

`ret = FusionRuntimeException.toString()`

Return the exception message.

`Return`

- `ret` (`string`) – The message.

13.2.7 Exception IOError

`mosek.fusion.IOError`

Error when reading or writing a stream, or opening a file.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.8 Exception IndexError

`mosek.fusion.IndexError`

Index out of bound, or a multi-dimensional index had wrong number of dimensions.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.9 Exception LengthError

`mosek.fusion.LengthError`

An array did not have the required length, or two arrays were expected to have same length.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.10 Exception MatrixError

`mosek.fusion.MatrixError`

Thrown if data used in construction of a matrix contained inconsistencies or errors.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.11 Exception ModelError

`mosek.fusion.ModelError`

Thrown when objects from different models were mixed.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.12 Exception NameError

`mosek.fusion.NameError`

Name clash; tries to add a variable or constraint with a name that already exists.

Members *FusionRuntimeException.toString* Return the exception message.

Implements

FusionRuntimeException

13.2.13 Exception OptimizeError

`mosek.fusion.OptimizeError`

An error occurred during optimization.

Members *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.14 Exception ParameterError

`mosek.fusion.ParameterError`
: Tried to use an invalid parameter for a value that was invalid for a specific parameter.

  `Members` *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.15 Exception RangeError

`mosek.fusion.RangeError`
: Invalid range specified

  `Members` *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.16 Exception SetDefinitionError

`mosek.fusion.SetDefinitionError`
: Invalid data for constructing set.

  `Members` *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.17 Exception SliceError

`mosek.fusion.SliceError`
: Invalid slice definition, negative slice or slice index out of bounds.

  `Members` *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.18 Exception SolutionError

`mosek.fusion.SolutionError`
: Requested a solution that was undefined or whose status was not acceptable.

  `Members` *FusionException.toString* Return the exception message.

Implements
: *FusionException*

13.2.19 Exception SparseFormatError

`mosek.fusion.SparseFormatError`
: The given sparsity patters was invalid or specified an index that was out of bounds.

  `Members` *FusionRuntimeException.toString* Return the exception message.

Implements
: *FusionRuntimeException*

13.2.20 Exception UnexpectedError

`mosek.fusion.UnexpectedError`

An unexpected error has happened. No specific excepion could have been risen.

Members `RuntimeException.toString` Return the exception message.

Implements

`RuntimeException`

13.2.21 Exception UnimplementedError

`mosek.fusion.UnimplementedError`

Called a stub. Functionality has not yet been implemented.

Members `RuntimeException.toString` Return the exception message.

Implements

`RuntimeException`

13.2.22 Exception ValueConversionError

`mosek.fusion.ValueConversionError`

Error casting or converting a value.

Members *`FusionRuntimeException.toString`* Return the exception message.

Implements

`FusionRuntimeException`

13.3 Enumerations

`AccSolutionStatus`

Constants used for defining which solutions statuses are acceptable.

`Anything`

Accept all solution status except *`Undefined`*.

`Optimal`

Accept only optimal solution status.

`NearOptimal`

Accept only optimal solution status.

`Feasible`

Accept any feasible solution, even if not optimal.

`Certificate`

Accept only a certificate.

`ObjectiveSense`

Used in *`Model.objective`* to define the objective sense of the *`Model`*.

`Undefined`

The sense is not defined; trying to optimize a *`Model`* whose objective sense is undefined is an error.

`Minimize`

Minimize the objective.

`Maximize`

Maximize the objective.

`PSDKey`

`IsSymPSD`

`IsTrilPSD`

`ProblemStatus`
Constants used for defining which solutions statuses are acceptable.

`Unknown`
Unknown problem status.

`PrimalAndDualFeasible`
The problem is feasible.

`PrimalFeasible`
The problem is at least primal feasible.

`DualFeasible`
The problem is at least least dual feasible.

`PrimalInfeasible`
The problem is primal infeasible.

`DualInfeasible`
The problem is dual infeasible.

`PrimalAndDualInfeasible`
The problem is primal and dual infeasible.

`IllPosed`
The problem is illposed.

`PrimalInfeasibleOrUnbounded`
The problem is primal infeasible or unbounded.

`QConeKey`

`InQCone`

`InRotatedQCone`

`RelationKey`
Used internally in *Fusion* to define the domain type for a constraint or variable.

`EqualsTo`

`LessThan`

`GreaterThan`

`IsFree`

`InRange`

`SolutionStatus`
Defines properties of either a primal or a dual solution. A model may contain multiple solutions which may have different status.

Specifically, there will be individual solutions, and thus solution statuses, for the interior-point, simplex and integer solvers.

`Undefined`
Undefined solution. This means that no values exist for the relevant solution.

`Unknown`
The solution status is unknown; this will happen if the user inputs values or a solution is read from a file.

`Optimal`
: The solution values are feasible and optimal.

`NearOptimal`
: The solution values are feasible and nearly optimal.

`Feasible`
: The solution is feasible.

`NearFeasible`
: The solution is nearly feasible.

`Certificate`
: The solution is a certificate of infeasibility (primal or dual, depending no which solution it belongs to).

`NearCertificate`
: The solution is nearly a certificate of infeasibility (primal or dual, depending no which solution it belongs to).

`IllposedCert`

`SolutionType`
Used when requesting a specific solution from a *Model*.

`Default`
: Auto-select the default solution; usually this will be the integer solution, if available, otherwise the basic solution, if available, otherwise the interior-point solution.

`Basic`
: Select the basic solution.

`Interior`
: Select the interior-point solution.

`Integer`
: Select the integer solution.

`StatusKey`
Defines the status of a single solution value.

`Unknown`
: The status is unknown; this will happen if, for example, the solution was read from a file or inputted by the user.

`Basic`
: The solution is basic.

`SuperBasic`
: The value is superbasic.

`OnBound`
: The value is on its bound.

`Infinity`
: The solution value is infinite, or sufficiently large to be deemed infinite.

13.4 Parameters

All parameters (alphabetical order)

Parameters grouped by topic

Note: some parameters may appear in more than one group.

- *Conic interior-point method*
- *Interior-point method*
- *Mixed-integer optimization*
- *Overall solver*
- *Solution input/output*
- *Termination criterion*
- *Analysis*
- *Optimization system*
- *Primal simplex optimizer*
- *Output information*
- *Basis identification*
- *Presolve*
- *Data input/output*
- *Infeasibility report*
- *Simplex optimizer*
- *Nonlinear convex method*
- *Dual simplex optimizer*
- *License manager*
- *Data check*
- *Logging*

13.4.1 Parameters List (alphabetically)

Double Parameters

`anaSolInfeasTol`

If a constraint violates its bound with an amount larger than this value, the constraint name, index and violation will be printed by the solution analyzer.

`Accepted Values:` [0.0 ;+inf]

`Default Value:` 1e-6

`Groups:` *Analysis*

`basisRelTolS`

Maximum relative dual bound violation allowed in an optimal basic solution.

`Accepted Values:` [0.0 ;+inf]

`Default Value:` 1.0e-12

`Groups:` *Simplex optimizer*, *Termination criterion*

`basisTolS`

Maximum absolute dual bound violation in an optimal basic solution.

`Accepted Values:` [1.0e-9 ;+inf]

`Default Value:` 1.0e-6

`Groups:` *Simplex optimizer*, *Termination criterion*

`basisTolX`
Maximum absolute primal bound violation allowed in an optimal basic solution.

Accepted Values: [1.0e-9 ;+inf]

Default Value: 1.0e-6

Groups: *Simplex optimizer*, *Termination criterion*

`intpntCoTolDfeas`
Dual feasibility tolerance used by the conic interior-point optimizer.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntCoTolInfeas`
Controls when the conic interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-10

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntCoTolMuRed`
Relative complementarity gap feasibility tolerance used by the conic interior-point optimizer.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntCoTolNearRel`
If **MOSEK** cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Accepted Values: [1.0 ;+inf]

Default Value: 1000

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntCoTolPfeas`
Primal feasibility tolerance used by the conic interior-point optimizer.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntCoTolRelGap`
Relative gap termination tolerance used by the conic interior-point optimizer.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-7

Groups: *Interior-point method*, *Termination criterion*, *Conic interior-point method*

`intpntQoTolDfeas`
Dual feasibility tolerance used when the interior-point optimizer is applied to a quadratic optimization problem..

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*

`intpntQoTolInfeas`
Controls when the conic interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-10

Groups: *Interior-point method*, *Termination criterion*

`intpntQoTolMuRed`
Relative complementarity gap feasibility tolerance used when interior-point optimizer is applied to a quadratic optimization problem.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*

`intpntQoTolNearRel`
If **MOSEK** cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Accepted Values: [1.0 ;+inf]

Default Value: 1000

Groups: *Interior-point method*, *Termination criterion*

`intpntQoTolPfeas`
Primal feasibility tolerance used when the interior-point optimizer is applied to a quadratic optimization problem.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*

`intpntQoTolRelGap`
Relative gap termination tolerance used when the interior-point optimizer is applied to a quadratic optimization problem.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*

`intpntTolDfeas`
Dual feasibility tolerance used for linear and quadratic optimization problems.

Accepted Values: [0.0 ;1.0]

Default Value: 1.0e-8

Groups: *Interior-point method*, *Termination criterion*

`intpntTolDsafe`
Controls the initial dual starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it might be worthwhile to increase this value.

Accepted Values: [1.0e-4 ;+inf]

Default Value: 1.0

Groups: *Interior-point method*

`intpntTolInfeas`

Controls when the optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible. A value of 0.0 means the optimizer must have an exact certificate of infeasibility and this is very unlikely to happen.

`Accepted Values:` [0.0 ;1.0]

`Default Value:` 1.0e-10

`Groups:` *Interior-point method*, *Termination criterion*, *Nonlinear convex method*

`intpntTolMuRed`

Relative complementarity gap tolerance.

`Accepted Values:` [0.0 ;1.0]

`Default Value:` 1.0e-16

`Groups:` *Interior-point method*, *Termination criterion*

`intpntTolPath`

Controls how close the interior-point optimizer follows the central path. A large value of this parameter means the central is followed very closely. On numerical unstable problems it may be worthwhile to increase this parameter.

`Accepted Values:` [0.0 ;0.9999]

`Default Value:` 1.0e-8

`Groups:` *Interior-point method*

`intpntTolPfeas`

Primal feasibility tolerance used for linear and quadratic optimization problems.

`Accepted Values:` [0.0 ;1.0]

`Default Value:` 1.0e-8

`Groups:` *Interior-point method*, *Termination criterion*

`intpntTolPsafe`

Controls the initial primal starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it may be worthwhile to increase this value.

`Accepted Values:` [1.0e-4 ;+inf]

`Default Value:` 1.0

`Groups:` *Interior-point method*

`intpntTolRelGap`

Relative gap termination tolerance.

`Accepted Values:` [1.0e-14 ;+inf]

`Default Value:` 1.0e-8

`Groups:` *Termination criterion*, *Interior-point method*

`intpntTolRelStep`

Relative step size to the boundary for linear and quadratic optimization problems.

`Accepted Values:` [1.0e-4 ;0.999999]

`Default Value:` 0.9999

`Groups:` *Interior-point method*

`intpntTolStepSize`

If the step size falls below the value of this parameter, then the interior-point optimizer assumes that it is stalled. In other words the interior-point optimizer does not make any progress and therefore it is better stop.

`Accepted Values:` [0.0 ;1.0]

`Default Value:` 1.0e-6

`Groups:` *Interior-point method*

`lowerObjCut`

If either a primal or dual feasible solution is found proving that the optimal objective value is outside, the interval [*lowerObjCut*, *upperObjCut*], then **MOSEK** is terminated.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1.0e30

`Groups:` *Termination criterion*

`lowerObjCutFiniteTrh`

If the lower objective cut is less than the value of this parameter value, then the lower objective cut i.e. *lowerObjCut* is treated as $-\infty$.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -0.5e30

`Groups:` *Termination criterion*

`mioDisableTermTime`

This parameter specifies the number of seconds n during which the termination criteria governed by

- *mioMaxNumRelaxs*
- *mioMaxNumBranches*
- *mioNearTolAbsGap*
- *mioNearTolRelGap*

is disabled since the beginning of the optimization.

A negative value is identical to infinity i.e. the termination criteria are never checked.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1.0

`Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioMaxTime`

This parameter limits the maximum time spent by the mixed-integer optimizer. A negative number means infinity.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1.0

`Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioNearTolAbsGap`

Relaxed absolute optimality tolerance employed by the mixed-integer optimizer. This termination criteria is delayed. See *mioDisableTermTime* for details.

`Accepted Values:` [0.0 ;+inf]

`Default Value:` 0.0

`Groups:` *Mixed-integer optimization*

`mioNearTolRelGap`
: The mixed-integer optimizer is terminated when this tolerance is satisfied. This termination criteria is delayed. See *mioDisableTermTime* for details.

  `Accepted Values:` [0.0 ;+inf]

  `Default Value:` 1.0e-3

  `Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioRelGapConst`
: This value is used to compute the relative gap for the solution to an integer optimization problem.

  `Accepted Values:` [1.0e-15 ;+inf]

  `Default Value:` 1.0e-10

  `Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioTolAbsGap`
: Absolute optimality tolerance employed by the mixed-integer optimizer.

  `Accepted Values:` [0.0 ;+inf]

  `Default Value:` 0.0

  `Groups:` *Mixed-integer optimization*

`mioTolAbsRelaxInt`
: Absolute relaxation tolerance of the integer constraints. I.e. $\min(|x| - \lfloor x \rfloor, \lceil x \rceil - |x|)$ is less than the tolerance then the integer restrictions assumed to be satisfied.

  `Accepted Values:` [1e-9 ;+inf]

  `Default Value:` 1.0e-5

  `Groups:` *Mixed-integer optimization*

`mioTolFeas`
: Feasibility tolerance for mixed integer solver.

  `Accepted Values:` [1e-9 ;1e-3]

  `Default Value:` 1.0e-6

  `Groups:` *Mixed-integer optimization*

`mioTolRelDualBoundImprovement`
: If the relative improvement of the dual bound is smaller than this value, the solver will terminate the root cut generation. A value of 0.0 means that the value is selected automatically.

  `Accepted Values:` [0.0 ;1.0]

  `Default Value:` 0.0

  `Groups:` *Mixed-integer optimization*

`mioTolRelGap`
: Relative optimality tolerance employed by the mixed-integer optimizer.

  `Accepted Values:` [0.0 ;+inf]

  `Default Value:` 1.0e-4

  `Groups:` *Mixed-integer optimization*, *Termination criterion*

`optimizerMaxTime`
: Maximum amount of time the optimizer is allowed to spent on the optimization. A negative number means infinity.

  `Accepted Values:` [-inf ;+inf]

  `Default Value:` -1.0

Groups: *Termination criterion*

`presolveTolAbsLindep`

Absolute tolerance employed by the linear dependency checker.

Accepted Values: [0.0 ;+inf]

Default Value: 1.0e-6

Groups: *Presolve*

`presolveTolAij`

Absolute zero tolerance employed for a_{ij} in the presolve.

Accepted Values: [1.0e-15 ;+inf]

Default Value: 1.0e-12

Groups: *Presolve*

`presolveTolRelLindep`

Relative tolerance employed by the linear dependency checker.

Accepted Values: [0.0 ;+inf]

Default Value: 1.0e-10

Groups: *Presolve*

`presolveTolS`

Absolute zero tolerance employed for s_i in the presolve.

Accepted Values: [0.0 ;+inf]

Default Value: 1.0e-8

Groups: *Presolve*

`presolveTolX`

Absolute zero tolerance employed for x_j in the presolve.

Accepted Values: [0.0 ;+inf]

Default Value: 1.0e-8

Groups: *Presolve*

`semidefiniteTolApprox`

Tolerance to define a matrix to be positive semidefinite.

Accepted Values: [1.0e-15 ;+inf]

Default Value: 1.0e-10

Groups: *Data check*

`simplexAbsTolPiv`

Absolute pivot tolerance employed by the simplex optimizers.

Accepted Values: [1.0e-12 ;+inf]

Default Value: 1.0e-7

Groups: *Simplex optimizer*

`simLuTolRelPiv`

Relative pivot tolerance employed when computing the LU factorization of the basis in the simplex optimizers and in the basis identification procedure.

A value closer to 1.0 generally improves numerical stability but typically also implies an increase in the computational work.

Accepted Values: [1.0e-6 ;0.999999]

Default Value: 0.01

Groups: *Basis identification*, *Simplex optimizer*

`upperObjCut`
If either a primal or dual feasible solution is found proving that the optimal objective value is outside, the interval [*lowerObjCut*, *upperObjCut*], then **MOSEK** is terminated.

Accepted Values: [-inf ;+inf]

Default Value: 1.0e30

Groups: *Termination criterion*

`upperObjCutFiniteTrh`
If the upper objective cut is greater than the value of this parameter, then the upper objective cut *upperObjCut* is treated as ∞.

Accepted Values: [-inf ;+inf]

Default Value: 0.5e30

Groups: *Termination criterion*

Integer Parameters

`autoUpdateSolInfo`
Controls whether the solution information items are automatically updated after an optimization is performed.

Accepted Values: ON, OFF

Default Value: off

Groups: *Optimization system*

`biCleanOptimizer`
Controls which simplex optimizer is used in the clean-up phase.

Accepted Values: FREE, INTPNT, CONIC, PRIMAL_SIMPLEX, DUAL_SIMPLEX, FREE_SIMPLEX, MIXED_INT

Default Value: free

Groups: *Basis identification*, *Overall solver*

`biIgnoreMaxIter`
If the parameter *intpntBasis* has the value `noError` and the interior-point optimizer has terminated due to maximum number of iterations, then basis identification is performed if this parameter has the value `on`.

Accepted Values: ON, OFF

Default Value: off

Groups: *Interior-point method*, *Basis identification*

`biIgnoreNumError`
If the parameter *intpntBasis* has the value `noError` and the interior-point optimizer has terminated due to a numerical problem, then basis identification is performed if this parameter has the value `on`.

Accepted Values: ON, OFF

Default Value: off

Groups: *Interior-point method*, *Basis identification*

`biMaxIterations`
Controls the maximum number of simplex iterations allowed to optimize a basis after the basis identification.

`Accepted Values:` [0 ;+inf]

`Default Value:` 1000000

`Groups:` *Basis identification*, *Termination criterion*

`cacheLicense`
Specifies if the license is kept checked out for the lifetime of the mosek environment (`on`) or returned to the server immediately after the optimization (`off`).

By default the license is checked out for the lifetime of the **MOSEK** environment by the first call to the optimizer.

Check-in and check-out of licenses have an overhead. Frequent communication with the license server should be avoided.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *License manager*

`infeasPreferPrimal`
If both certificates of primal and dual infeasibility are supplied then only the primal is used when this option is turned on.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Overall solver*

`intpntBasis`
Controls whether the interior-point optimizer also computes an optimal basis.

`Accepted Values:` NEVER, ALWAYS, NO_ERROR, IF_FEASIBLE, RESERVERED

`Default Value:` always

`Groups:` *Interior-point method*, *Basis identification*

`intpntDiffStep`
Controls whether different step sizes are allowed in the primal and dual space.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Interior-point method*

`intpntHotstart`
Currently not in use.

`Accepted Values:` NONE, PRIMAL, DUAL, PRIMAL_DUAL

`Default Value:` none

`Groups:` *Interior-point method*

`intpntMaxIterations`
Controls the maximum number of iterations allowed in the interior-point optimizer.

`Accepted Values:` [0 ;+inf]

`Default Value:` 400

`Groups:` *Interior-point method*, *Termination criterion* `intpntMaxNumCor`

Controls the maximum number of correctors allowed by the multiple corrector procedure. A negative value means that **MOSEK** is making the choice.

`Accepted Values:` [-1 ;+inf]

`Default Value:` -1

`Groups:` *Interior-point method*

`intpntMaxNumRefinementSteps`

Maximum number of steps to be used by the iterative refinement of the search direction. A negative value implies that the optimizer chooses the maximum number of iterative refinement steps.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Interior-point method*

`intpntMultiThread`

Controls whether the interior-point optimizers are allowed to employ multiple threads if more threads is available.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Optimization system*

`intpntOffColTrh`

Controls how many offending columns are detected in the Jacobian of the constraint matrix.

0	no detection
1	aggressive detection
> 1	higher values mean less aggressive detection

`Accepted Values:` [0 ;+inf]

`Default Value:` 40

`Groups:` *Interior-point method*

`intpntOrderMethod`

Controls the ordering strategy used by the interior-point optimizer when factorizing the Newton equation system.

`Accepted Values:` FREE, APPMINLOC, EXPERIMENTAL, TRY_GRAPHPAR, FORCE_GRAPHPAR, NONE

`Default Value:` free

`Groups:` *Interior-point method*

`intpntRegularizationUse`

Controls whether regularization is allowed.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Interior-point method*

`intpntScaling`

Controls how the problem is scaled before the interior-point optimizer is used.

`Accepted Values:` FREE, NONE, MODERATE, AGGRESSIVE

`Default Value:` free

`Groups:` *Interior-point method*

`intpntSolveForm`
Controls whether the primal or the dual problem is solved.

`Accepted Values:` FREE, PRIMAL, DUAL

`Default Value:` free

`Groups:` *Interior-point method*

`intpntStartingPoint`
Starting point used by the interior-point optimizer.

`Accepted Values:` FREE, GUESS, CONSTANT, SATISFY_BOUNDS

`Default Value:` free

`Groups:` *Interior-point method*

`licenseDebug`
This option is used to turn on debugging of the license manager.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *License manager*

`licensePauseTime`
If *`licenseWait`* = `on` and no license is available, then **MOSEK** sleeps a number of milliseconds between each check of whether a license has become free.

`Accepted Values:` [0 ;1000000]

`Default Value:` 100

`Groups:` *License manager*

`licenseSuppressExpireWrns`
Controls whether license features expire warnings are suppressed.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *License manager*, *Output information*

`licenseTrhExpiryWrn`
If a license feature expires in a numbers days less than the value of this parameter then a warning will be issued.

`Accepted Values:` [0 ;+inf]

`Default Value:` 7

`licenseWait`
If all licenses are in use **MOSEK** returns with an error code. However, by turning on this parameter **MOSEK** will wait for an available license.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *Overall solver*, *Optimization system*, *License manager*

`log`
Controls the amount of log information. The value 0 implies that all log information is suppressed. A higher level implies that more information is logged.

Please note that if a task is employed to solve a sequence of optimization problems the value of this parameter is reduced by the value of *`logCutSecondOpt`* for the second and any subsequent optimizations.

`Accepted Values:` [0 ;+inf]

Default Value: 10

Groups: *Output information*, *Logging*

`logAnaPro`

Controls amount of output from the problem analyzer.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Analysis*, *Logging*

`logBi`

Controls the amount of output printed by the basis identification procedure. A higher level implies that more information is logged.

Accepted Values: [0 ;+inf]

Default Value: 4

Groups: *Basis identification*, *Output information*, *Logging*

`logBiFreq`

Controls how frequent the optimizer outputs information about the basis identification and how frequent the user-defined call-back function is called.

Accepted Values: [0 ;+inf]

Default Value: 2500

Groups: *Basis identification*, *Output information*, *Logging*

`logCutSecondOpt`

If a task is employed to solve a sequence of optimization problems, then the value of the log levels is reduced by the value of this parameter. E.g `log` and `logSim` are reduced by the value of this parameter for the second and any subsequent optimizations.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Output information*, *Logging*

`logExpand`

Controls the amount of logging when a data item such as the maximum number constrains is expanded.

Accepted Values: [0 ;+inf]

Default Value: 0

Groups: *Output information*, *Logging*

`logFactor`

If turned on, then the factor log lines are added to the log.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Output information*, *Logging*

`logFile`

If turned on, then some log info is printed when a file is written or read.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Data input/output*, *Output information*, *Logging* `logHead`

If turned on, then a header line is added to the log.

`Accepted Values:` [0 ;+inf]

`Default Value:` 1

`Groups:` *Output information*, *Logging*

`logInfeasAna`

Controls amount of output printed by the infeasibility analyzer procedures. A higher level implies that more information is logged.

`Accepted Values:` [0 ;+inf]

`Default Value:` 1

`Groups:` *Infeasibility report*, *Output information*, *Logging*

`logIntpnt`

Controls amount of output printed by the interior-point optimizer. A higher level implies that more information is logged.

`Accepted Values:` [0 ;+inf]

`Default Value:` 4

`Groups:` *Interior-point method*, *Output information*, *Logging*

`logMio`

Controls the log level for the mixed-integer optimizer. A higher level implies that more information is logged.

`Accepted Values:` [0 ;+inf]

`Default Value:` 4

`Groups:` *Mixed-integer optimization*, *Output information*, *Logging*

`logMioFreq`

Controls how frequent the mixed-integer optimizer prints the log line. It will print line every time *`logMioFreq`* relaxations have been solved.

`Accepted Values:` [-inf ;+inf]

`Default Value:` 10

`Groups:` *Mixed-integer optimization*, *Output information*, *Logging*

`logOptimizer`

Controls the amount of general optimizer information that is logged.

`Accepted Values:` [0 ;+inf]

`Default Value:` 1

`Groups:` *Output information*, *Logging*

`logOrder`

If turned on, then factor lines are added to the log.

`Accepted Values:` [0 ;+inf]

`Default Value:` 1

`Groups:` *Output information*, *Logging*

`logPresolve`

Controls amount of output printed by the presolve procedure. A higher level implies that more information is logged.

`Accepted Values:` [0 ;+inf]

Default Value: 1

Groups: *Interior-point method*, *Logging*

logResponse
Controls amount of output printed when response codes are reported. A higher level implies that more information is logged.

Accepted Values: [0 ;+inf]

Default Value: 0

Groups: *Output information*, *Logging*

logSim
Controls amount of output printed by the simplex optimizer. A higher level implies that more information is logged.

Accepted Values: [0 ;+inf]

Default Value: 4

Groups: *Simplex optimizer*, *Output information*, *Logging*

logSimFreq
Controls how frequent the simplex optimizer outputs information about the optimization and how frequent the user-defined call-back function is called.

Accepted Values: [0 ;+inf]

Default Value: 1000

Groups: *Simplex optimizer*, *Output information*, *Logging*

logSimMinor
Currently not in use.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Simplex optimizer*, *Output information*

logStorage
When turned on, **MOSEK** prints messages regarding the storage usage and allocation.

Accepted Values: [0 ;+inf]

Default Value: 0

Groups: *Output information*, *Optimization system*, *Logging*

maxNumWarnings
Each warning is shown a limit number times controlled by this parameter. A negative value is identical to infinite number of times.

Accepted Values: [-inf ;+inf]

Default Value: 10

Groups: *Output information*

mioBranchDir
Controls whether the mixed-integer optimizer is branching up or down by default.

Accepted Values: FREE, UP, DOWN, NEAR, FAR, ROOT_LP, GUIDED, PSEUDOCOST

Default Value: free

Groups: *Mixed-integer optimization*

`mioConstructSol`
If set to `on` and all integer variables have been given a value for which a feasible mixed integer solution exists, then **MOSEK** generates an initial solution to the mixed integer problem by fixing all integer values and solving the remaining problem.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *Mixed-integer optimization*

`mioCutClique`
Controls whether clique cuts should be generated.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Mixed-integer optimization*

`mioCutCmir`
Controls whether mixed integer rounding cuts should be generated.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Mixed-integer optimization*

`mioCutGmi`
Controls whether GMI cuts should be generated.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Mixed-integer optimization*

`mioCutImpliedBound`
Controls whether implied bound cuts should be generated.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *Mixed-integer optimization*

`mioCutKnapsackCover`
Controls whether knapsack cover cuts should be generated.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *Mixed-integer optimization*

`mioCutSelectionLevel`
Controls how aggressively generated cuts are selected to be included in the relaxation.

-1. The optimizer chooses the level of cut selection

0.Generated cuts less likely to be added to the relaxation

1.Cuts are more aggressively selected to be included in the relaxation

`Accepted Values:` [-1 ;+1]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*

`mioHeuristicLevel`

Controls the heuristic employed by the mixed-integer optimizer to locate an initial good integer feasible solution. A value of zero means the heuristic is not used at all. A larger value than 0 means that a gradually more sophisticated heuristic is used which is computationally more expensive. A negative value implies that the optimizer chooses the heuristic. Normally a value around 3 to 5 should be optimal.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*

`mioMaxNumBranches`

Maximum number of branches allowed during the branch and bound search. A negative value means infinite.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioMaxNumRelaxs`

Maximum number of relaxations allowed during the branch and bound search. A negative value means infinite.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*

`mioMaxNumSolutions`

The mixed-integer optimizer can be terminated after a certain number of different feasible solutions has been located. If this parameter has the value $n > 0$, then the mixed-integer optimizer will be terminated when n feasible solutions have been located.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*, *Termination criterion*

`mioMode`

Controls whether the optimizer includes the integer restrictions when solving a (mixed) integer optimization problem.

`Accepted Values:` IGNORED, SATISFIED

`Default Value:` satisfied

`Groups:` *Overall solver*

`mioMtUserCb`

It true user callbacks are called from each thread used by this optimizer. If false the user callback is only called from a single thread.

`Accepted Values:` ON, OFF

`Default Value:` off

`Groups:` *Optimization system*

`mioNodeOptimizer`

Controls which optimizer is employed at the non-root nodes in the mixed-integer optimizer.

`Accepted Values:` FREE, INTPNT, CONIC, PRIMAL_SIMPLEX, DUAL_SIMPLEX, FREE_SIMPLEX, MIXED_INT

`Default Value:` free

Groups: *Mixed-integer optimization*

mioNodeSelection
Controls the node selection strategy employed by the mixed-integer optimizer.

Accepted Values: FREE, FIRST, BEST, WORST, HYBRID, PSEUDO

Default Value: free

Groups: *Mixed-integer optimization*

mioPerspectiveReformulate
Enables or disables perspective reformulation in presolve.

Accepted Values: ON, OFF

Default Value: on

Groups: *Mixed-integer optimization*

mioProbingLevel
Controls the amount of probing employed by the mixed-integer optimizer in presolve.

-1. The optimizer chooses the level of probing employed

0.Probing is disabled

1.A low amount of probing is employed

2.A medium amount of probing is employed

3.A high amount of probing is employed

Accepted Values: [-inf ;+inf]

Default Value: -1

Groups: *Mixed-integer optimization*

mioRinsMaxNodes
Controls the maximum number of nodes allowed in each call to the RINS heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Accepted Values: [-1 ;+inf]

Default Value: -1

Groups: *Mixed-integer optimization*

mioRootOptimizer
Controls which optimizer is employed at the root node in the mixed-integer optimizer.

Accepted Values: FREE, INTPNT, CONIC, PRIMAL_SIMPLEX, DUAL_SIMPLEX, FREE_SIMPLEX, MIXED_INT

Default Value: free

Groups: *Mixed-integer optimization*

mioRootRepeatPresolveLevel
Controls whether presolve can be repeated at root node.

- -1 The optimizer chooses whether presolve is repeated
- 0 Never repeat presolve
- 1 Always repeat presolve

Accepted Values: [-1 ;1]

Default Value: -1

Groups: *Mixed-integer optimization*

`mioVbDetectionLevel`
Controls how much effort is put into detecting variable bounds.

-1. The optimizer chooses

0.No variable bounds are detected

1.Only detect variable bounds that are directly represented in the problem

2.Detect variable bounds in probing

`Accepted Values:` [-1 ;+2]

`Default Value:` -1

`Groups:` *Mixed-integer optimization*

`mtSpincount`
Set the number of iterations to spin before sleeping.

`Accepted Values:` [0 ;1000000000]

`Default Value:` 0

`Groups:` *Optimization system*

`numThreads`
Controls the number of threads employed by the optimizer. If set to 0 the number of threads used will be equal to the number of cores detected on the machine.

`Accepted Values:` [0 ;+inf]

`Default Value:` 0

`Groups:` *Optimization system*

`optimizer`
The parameter controls which optimizer is used to optimize the task.

`Accepted Values:` FREE, INTPNT, CONIC, PRIMAL_SIMPLEX, DUAL_SIMPLEX, FREE_SIMPLEX, MIXED_INT

`Default Value:` free

`Groups:` *Overall solver*

`presolveEliminatorMaxFill`
Controls the maximum amount of fill-in that can be created by one pivot in the elimination phase of the presolve. A negative value menas the parameter value is selected automatically.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Presolve*

`presolveEliminatorMaxNumTries`
Control the maximum number of times the eliminator is tried.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Presolve*

`presolveLevel`
Currently not used.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`Groups:` *Overall solver*, *Presolve*

`presolveLindepAbsWorkTrh`

The linear dependency check is potentially computationally expensive.

`Accepted Values:` [-inf ;+inf]

`Default Value:` 100

`Groups:` *Presolve*

`presolveLindepRelWorkTrh`

The linear dependency check is potentially computationally expensive.

`Accepted Values:` [-inf ;+inf]

`Default Value:` 100

`Groups:` *Presolve*

`presolveLindepUse`

Controls whether the linear constraints are checked for linear dependencies.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Presolve*

`presolveMaxNumReductions`

Controls the maximum number of reductions performed by the presolve. The value of the parameter is normally only changed in connection with debugging. A negative value implies that an infinite number of reductions are allowed.

`Accepted Values:` [-inf ;+inf]

`Default Value:` -1

`presolveUse`

Controls whether the presolve is applied to a problem before it is optimized.

`Accepted Values:` OFF, ON, FREE

`Default Value:` free

`Groups:` *Overall solver*, *Presolve*

`simBasisFactorUse`

Controls whether a (LU) factorization of the basis is used in a hot-start. Forcing a refactorization sometimes improves the stability of the simplex optimizers, but in most cases there is a performance penalty.

`Accepted Values:` ON, OFF

`Default Value:` on

`Groups:` *Simplex optimizer*

`simDegen`

Controls how aggressively degeneration is handled.

`Accepted Values:` NONE, FREE, AGGRESSIVE, MODERATE, MINIMUM

`Default Value:` free

`Groups:` *Simplex optimizer*

`simDualCrash`

Controls whether crashing is performed in the dual simplex optimizer.

If this parameter is set to x, then a crash will be performed if a basis consists of more than $(100-x)$ mod f_v entries, where f_v is the number of fixed variables.

`Accepted Values:` [0 ;+inf]

Default Value: 90

Groups: *Dual simplex optimizer*

simDualPhaseoneMethod
An experimental feature.

Accepted Values: [0 ;10]

Default Value: 0

Groups: *Simplex optimizer*

simDualRestrictSelection
The dual simplex optimizer can use a so-called restricted selection/pricing strategy to chooses the outgoing variable. Hence, if restricted selection is applied, then the dual simplex optimizer first choose a subset of all the potential outgoing variables. Next, for some time it will choose the outgoing variable only among the subset. From time to time the subset is redefined.

A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Accepted Values: [0 ;100]

Default Value: 50

Groups: *Dual simplex optimizer*

simDualSelection
Controls the choice of the incoming variable, known as the selection strategy, in the dual simplex optimizer.

Accepted Values: FREE, FULL, ASE, DEVEX, SE, PARTIAL

Default Value: free

Groups: *Dual simplex optimizer*

simExploitDupvec
Controls if the simplex optimizers are allowed to exploit duplicated columns.

Accepted Values: ON, OFF, FREE

Default Value: off

Groups: *Simplex optimizer*

simHotstart
Controls the type of hot-start that the simplex optimizer perform.

Accepted Values: NONE, FREE, STATUS_KEYS

Default Value: free

Groups: *Simplex optimizer*

simHotstartLu
Determines if the simplex optimizer should exploit the initial factorization.

Accepted Values: ON, OFF

Default Value: on

simInteger
An experimental feature.

Accepted Values: [0 ;10]

Default Value: 0

Groups: *Simplex optimizer*

simMaxIterations
Maximum number of iterations that can be used by a simplex optimizer.

Accepted Values: [0 ;+inf]

Default Value: 10000000

Groups: *Simplex optimizer*, *Termination criterion*

simMaxNumSetbacks
Controls how many set-backs are allowed within a simplex optimizer. A set-back is an event where the optimizer moves in the wrong direction. This is impossible in theory but may happen due to numerical problems.

Accepted Values: [0 ;+inf]

Default Value: 250

Groups: *Simplex optimizer*

simNonSingular
Controls if the simplex optimizer ensures a non-singular basis, if possible.

Accepted Values: ON, OFF

Default Value: on

Groups: *Simplex optimizer*

simPrimalCrash
Controls whether crashing is performed in the primal simplex optimizer.

In general, if a basis consists of more than (100-this parameter value)% fixed variables, then a crash will be performed.

Accepted Values: [0 ;+inf]

Default Value: 90

Groups: *Primal simplex optimizer*

simPrimalPhaseoneMethod
An experimental feature.

Accepted Values: [0 ;10]

Default Value: 0

Groups: *Simplex optimizer*

simPrimalRestrictSelection
The primal simplex optimizer can use a so-called restricted selection/pricing strategy to chooses the outgoing variable. Hence, if restricted selection is applied, then the primal simplex optimizer first choose a subset of all the potential incoming variables. Next, for some time it will choose the incoming variable only among the subset. From time to time the subset is redefined.

A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Accepted Values: [0 ;100]

Default Value: 50

Groups: *Primal simplex optimizer*

simPrimalSelection
Controls the choice of the incoming variable, known as the selection strategy, in the primal simplex optimizer.

Accepted Values: FREE, FULL, ASE, DEVEX, SE, PARTIAL

Default Value: free

Groups: *Primal simplex optimizer*

simRefactorFreq
Controls how frequent the basis is refactorized. The value 0 means that the optimizer determines the best point of refactorization.

It is strongly recommended NOT to change this parameter.

Accepted Values: [0 ;+inf]

Default Value: 0

Groups: *Simplex optimizer*

simReformulation
Controls if the simplex optimizers are allowed to reformulate the problem.

Accepted Values: ON, OFF, FREE, AGGRESSIVE

Default Value: off

Groups: *Simplex optimizer*

simSaveLu
Controls if the LU factorization stored should be replaced with the LU factorization corresponding to the initial basis.

Accepted Values: ON, OFF

Default Value: off

Groups: *Simplex optimizer*

simScaling
Controls how much effort is used in scaling the problem before a simplex optimizer is used.

Accepted Values: FREE, NONE, MODERATE, AGGRESSIVE

Default Value: free

Groups: *Simplex optimizer*

simScalingMethod
Controls how the problem is scaled before a simplex optimizer is used.

Accepted Values: POW2, FREE

Default Value: pow2

Groups: *Simplex optimizer*

simSolveForm
Controls whether the primal or the dual problem is solved by the primal-/dual-simplex optimizer.

Accepted Values: FREE, PRIMAL, DUAL

Default Value: free

Groups: *Simplex optimizer*

simStabilityPriority
Controls how high priority the numerical stability should be given.

Accepted Values: [0 ;100]

Default Value: 50

Groups: *Simplex optimizer*

simSwitchOptimizer
The simplex optimizer sometimes chooses to solve the dual problem instead of the primal problem. This implies that if you have chosen to use the dual simplex optimizer and the problem is dualized, then it actually makes sense to use the primal simplex optimizer instead. If this parameter is on

and the problem is dualized and furthermore the simplex optimizer is chosen to be the primal (dual) one, then it is switched to the dual (primal).

Accepted Values: ON, OFF

Default Value: off

Groups: *Simplex optimizer*

`timingLevel`
Controls the a amount of timing performed inside **MOSEK**.

Accepted Values: [0 ;+inf]

Default Value: 1

Groups: *Optimization system*

`writeLpFullObj`
Write all variables, including the ones with 0-coefficients, in the objective.

Accepted Values: ON, OFF

Default Value: on

Groups: *Data input/output*

`writeLpLineWidth`
Maximum width of line in an LP file written by **MOSEK**.

Accepted Values: [40 ;+inf]

Default Value: 80

Groups: *Data input/output*

`writeLpQuotedNames`
If this option is turned on, then **MOSEK** will quote invalid LP names when writing an LP file.

Accepted Values: ON, OFF

Default Value: on

Groups: *Data input/output*

`writeLpTermsPerLine`
Maximum number of terms on a single line in an LP file written by **MOSEK**. 0 means unlimited.

Accepted Values: [0 ;+inf]

Default Value: 10

Groups: *Data input/output*

String Parameters

`basSolFileName`
Name of the `bas` solution file.

Accepted Values: Any valid file name.

Groups: *Data input/output*, *Solution input/output*

`dataFileName`
Data are read and written to this file.

Accepted Values: Any valid file name.

Groups: *Data input/output*

`debugFileName`

MOSEK debug file.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*

`intSolFileName`

Name of the `int` solution file.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*, *Solution input/output*

`itrSolFileName`

Name of the `itr` solution file.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*, *Solution input/output*

`mioDebugString`

For internal use only.

`Accepted Values:` Any valid string.

`Groups:` *Data input/output*

`paramCommentSign`

Only the first character in this string is used. It is considered as a start of comment sign in the **MOSEK** parameter file. Spaces are ignored in the string.

`Accepted Values:` Any valid string.

`Default Value:` %%

`Groups:` *Data input/output*

`paramReadFileName`

Modifications to the parameter database is read from this file.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*

`paramWriteFileName`

The parameter database is written to this file.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*

`readMpsBouName`

Name of the BOUNDS vector used. An empty name means that the first BOUNDS vector is used.

`Accepted Values:` Any valid MPS name.

`Groups:` *Data input/output*

`readMpsObjName`

Name of the free constraint used as objective function. An empty name means that the first constraint is used as objective function.

`Accepted Values:` Any valid MPS name.

`Groups:` *Data input/output*

`readMpsRanName`

Name of the RANGE vector used. An empty name means that the first RANGE vector is used.

`Accepted Values:` Any valid MPS name.

`Groups:` *Data input/output*

`readMpsRhsName`
Name of the RHS used. An empty name means that the first RHS vector is used.

`Accepted Values:` Any valid MPS name.

`Groups:` *Data input/output*

`remoteAccessToken`
An access token used to submit tasks to a remote **MOSEK** server. An access token is a random 32-byte string encoded in base64, i.e. it is a 44 character ASCII string.

`Accepted Values:` Any valid string.

`solFilterXcLow`
A filter used to determine which constraints should be listed in the solution file. A value of 0.5 means that all constraints having `xc[i]>0.5` should be listed, whereas `+0.5` means that all constraints having `xc[i]>=blc[i]+0.5` should be listed. An empty filter means that no filter is applied.

`Accepted Values:` Any valid filter.

`Groups:` *Data input/output*, *Solution input/output*

`solFilterXcUpr`
A filter used to determine which constraints should be listed in the solution file. A value of `0.5` means that all constraints having `xc[i]<0.5` should be listed, whereas `-0.5` means all constraints having `xc[i]<=buc[i]-0.5` should be listed. An empty filter means that no filter is applied.

`Accepted Values:` Any valid filter.

`Groups:` *Data input/output*, *Solution input/output*

`solFilterXxLow`
A filter used to determine which variables should be listed in the solution file. A value of "0.5" means that all constraints having `xx[j]>=0.5` should be listed, whereas "+0.5" means that all constraints having `xx[j]>=blx[j]+0.5` should be listed. An empty filter means no filter is applied.

`Accepted Values:` Any valid filter.

`Groups:` *Data input/output*, *Solution input/output*

`solFilterXxUpr`
A filter used to determine which variables should be listed in the solution file. A value of "0.5" means that all constraints having `xx[j]<0.5` should be printed, whereas "-0.5" means all constraints having `xx[j]<=bux[j]-0.5` should be listed. An empty filter means no filter is applied.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*, *Solution input/output*

`statFileName`
Statistics file name.

`Accepted Values:` Any valid file name.

`Groups:` *Data input/output*

`statKey`
Key used when writing the summary file.

`Accepted Values:` Any valid XML string.

`Groups:` *Data input/output*

`statName`
Name used when writing the statistics file.

`Accepted Values:` Any valid XML string.

`Groups:` *Data input/output*

`writeLpGenVarName`

Sometimes when an LP file is written additional variables must be inserted. They will have the prefix denoted by this parameter.

`Accepted Values:` Any valid string.

`Default Value:` xmskgen

`Groups:` *Data input/output*

13.4.2 Conic interior-point method parameters.

- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*
- *intpntCoTolPfeas*
- *intpntCoTolRelGap*

13.4.3 Interior-point method parameters.

- *biIgnoreMaxIter*
- *biIgnoreNumError*
- *intpntBasis*
- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*
- *intpntCoTolPfeas*
- *intpntCoTolRelGap*
- *intpntDiffStep*
- *intpntHotstart*
- *intpntMaxIterations*
- *intpntMaxNumCor*
- *intpntMaxNumRefinementSteps*
- *intpntOffColTrh*
- *intpntOrderMethod*
- *intpntQoTolDfeas*
- *intpntQoTolInfeas*
- *intpntQoTolMuRed*
- *intpntQoTolNearRel*
- *intpntQoTolPfeas*
- *intpntQoTolRelGap*
- *intpntRegularizationUse*

- *intpntScaling*
- *intpntSolveForm*
- *intpntStartingPoint*
- *intpntTolDfeas*
- *intpntTolDsafe*
- *intpntTolInfeas*
- *intpntTolMuRed*
- *intpntTolPath*
- *intpntTolPfeas*
- *intpntTolPsafe*
- *intpntTolRelGap*
- *intpntTolRelStep*
- *intpntTolStepSize*
- *logIntpnt*
- *logPresolve*

13.4.4 Mixed-integer optimization parameters.

- *logMio*
- *logMioFreq*
- *mioBranchDir*
- *mioConstructSol*
- *mioCutClique*
- *mioCutCmir*
- *mioCutGmi*
- *mioCutImpliedBound*
- *mioCutKnapsackCover*
- *mioCutSelectionLevel*
- *mioDisableTermTime*
- *mioHeuristicLevel*
- *mioMaxNumBranches*
- *mioMaxNumRelaxs*
- *mioMaxNumSolutions*
- *mioMaxTime*
- *mioNearTolAbsGap*
- *mioNearTolRelGap*
- *mioNodeOptimizer*
- *mioNodeSelection*
- *mioPerspectiveReformulate*
- *mioProbingLevel*

- *mioRelGapConst*
- *mioRinsMaxNodes*
- *mioRootOptimizer*
- *mioRootRepeatPresolveLevel*
- *mioTolAbsGap*
- *mioTolAbsRelaxInt*
- *mioTolFeas*
- *mioTolRelDualBoundImprovement*
- *mioTolRelGap*
- *mioVbDetectionLevel*

13.4.5 Overall solver parameters.

- *biCleanOptimizer*
- *infeasPreferPrimal*
- *licenseWait*
- *mioMode*
- *optimizer*
- *presolveLevel*
- *presolveUse*

13.4.6 Solution input/output parameters.

- *basSolFileName*
- *intSolFileName*
- *itrSolFileName*
- *solFilterXcLow*
- *solFilterXcUpr*
- *solFilterXxLow*
- *solFilterXxUpr*

13.4.7 Termination criterion parameters.

- *basisRelTolS*
- *basisTolS*
- *basisTolX*
- *biMaxIterations*
- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*

- *intpntCoTolPfeas*
- *intpntCoTolRelGap*
- *intpntMaxIterations*
- *intpntQoTolDfeas*
- *intpntQoTolInfeas*
- *intpntQoTolMuRed*
- *intpntQoTolNearRel*
- *intpntQoTolPfeas*
- *intpntQoTolRelGap*
- *intpntTolDfeas*
- *intpntTolInfeas*
- *intpntTolMuRed*
- *intpntTolPfeas*
- *intpntTolRelGap*
- *lowerObjCut*
- *lowerObjCutFiniteTrh*
- *mioDisableTermTime*
- *mioMaxNumBranches*
- *mioMaxNumSolutions*
- *mioMaxTime*
- *mioNearTolRelGap*
- *mioRelGapConst*
- *mioTolRelGap*
- *optimizerMaxTime*
- *simMaxIterations*
- *upperObjCut*
- *upperObjCutFiniteTrh*

13.4.8 Analysis parameters.

- *anaSolInfeasTol*
- *logAnaPro*

13.4.9 Optimization system parameters.

- *autoUpdateSolInfo*
- *intpntMultiThread*
- *licenseWait*
- *logStorage*
- *mioMtUserCb*

- *mtSpincount*
- *numThreads*
- *timingLevel*

13.4.10 Primal simplex optimizer parameters.

- *simPrimalCrash*
- *simPrimalRestrictSelection*
- *simPrimalSelection*

13.4.11 Output information parameters.

- *licenseSuppressExpireWrns*
- *log*
- *logBi*
- *logBiFreq*
- *logCutSecondOpt*
- *logExpand*
- *logFactor*
- *logFile*
- *logHead*
- *logInfeasAna*
- *logIntpnt*
- *logMio*
- *logMioFreq*
- *logOptimizer*
- *logOrder*
- *logResponse*
- *logSim*
- *logSimFreq*
- *logSimMinor*
- *logStorage*
- *maxNumWarnings*

13.4.12 Basis identification parameters.

- *biCleanOptimizer*
- *biIgnoreMaxIter*
- *biIgnoreNumError*
- *biMaxIterations*
- *intpntBasis*

- *logBi*
- *logBiFreq*
- *simLuTolRelPiv*

13.4.13 Presolve parameters.

- *presolveEliminatorMaxFill*
- *presolveEliminatorMaxNumTries*
- *presolveLevel*
- *presolveLindepAbsWorkTrh*
- *presolveLindepRelWorkTrh*
- *presolveLindepUse*
- *presolveTolAbsLindep*
- *presolveTolAij*
- *presolveTolRelLindep*
- *presolveTolS*
- *presolveTolX*
- *presolveUse*

13.4.14 Data input/output parameters.

- *basSolFileName*
- *dataFileName*
- *debugFileName*
- *intSolFileName*
- *itrSolFileName*
- *logFile*
- *mioDebugString*
- *paramCommentSign*
- *paramReadFileName*
- *paramWriteFileName*
- *readMpsBouName*
- *readMpsObjName*
- *readMpsRanName*
- *readMpsRhsName*
- *solFilterXcLow*
- *solFilterXcUpr*
- *solFilterXxLow*
- *solFilterXxUpr*
- *statFileName*

- *statKey*
- *statName*
- *writeLpFullObj*
- *writeLpGenVarName*
- *writeLpLineWidth*
- *writeLpQuotedNames*
- *writeLpTermsPerLine*

13.4.15 Infeasibility report parameters.

- *logInfeasAna*

13.4.16 Simplex optimizer parameters.

- *basisRelTolS*
- *basisTolS*
- *basisTolX*
- *logSim*
- *logSimFreq*
- *logSimMinor*
- *simBasisFactorUse*
- *simDegen*
- *simDualPhaseoneMethod*
- *simExploitDupvec*
- *simHotstart*
- *simInteger*
- *simLuTolRelPiv*
- *simMaxIterations*
- *simMaxNumSetbacks*
- *simNonSingular*
- *simPrimalPhaseoneMethod*
- *simRefactorFreq*
- *simReformulation*
- *simSaveLu*
- *simScaling*
- *simScalingMethod*
- *simSolveForm*
- *simStabilityPriority*
- *simSwitchOptimizer*
- *simplexAbsTolPiv*

13.4.17 Nonlinear convex method parameters.

- *intpntTolInfeas*

13.4.18 Dual simplex optimizer parameters.

- *simDualCrash*
- *simDualRestrictSelection*
- *simDualSelection*

13.4.19 License manager parameters.

- *cacheLicense*
- *licenseDebug*
- *licensePauseTime*
- *licenseSuppressExpireWrns*
- *licenseWait*

13.4.20 Data check parameters.

- *semidefiniteTolApprox*

13.4.21 Logging parameters.

- *log*
- *logAnaPro*
- *logBi*
- *logBiFreq*
- *logCutSecondOpt*
- *logExpand*
- *logFactor*
- *logFile*
- *logHead*
- *logInfeasAna*
- *logIntpnt*
- *logMio*
- *logMioFreq*
- *logOptimizer*
- *logOrder*
- *logPresolve*
- *logResponse*
- *logSim*
- *logSimFreq*

- *logStorage*

CHAPTER

FOURTEEN

SUPPORTED FILE FORMATS

MOSEK supports a range of problem and solution formats listed in Table 14.1 and Table 14.2. The **Task format** is **MOSEK**'s native binary format and it supports all features that **MOSEK** supports. The **OPF format** is **MOSEK**'s human-readable alternative that supports nearly all features (everything except semidefinite problems). In general, text formats are significantly slower to read, but can be examined and edited directly in any text editor.

Problem formats

See Table 14.1.

Table 14.1: List of supported file formats for optimization problems.

Format Type	Ext.	Binary/Text	LP	QP	CQO	SDP
LP	lp	plain text	X	X		
MPS	mps	plain text	X	X		
OPF	opf	plain text	X	X	X	
CBF	cbf	plain text	X		X	X
Osil	xml	xml text	X	X		
Task format	task	binary	X	X	X	X
Jtask format	jtask	text	X	X	X	X

Solution formats

See Table 14.2.

Table 14.2: List of supported solution formats.

Format Type	Ext.	Binary/Text	Description
SOL	sol	plain text	Interior Solution
	bas	plain text	Basic Solution
	int	plain text	Integer
Jsol format	jsol	text	Solution

Compression

MOSEK supports GZIP compression of files. Problem files with an additional `.gz` extension are assumed to be compressed when read, and are automatically compressed when written. For example, a file called

```
problem.mps.gz
```

will be considered as a GZIP compressed MPS file.

14.1 The LP File Format

MOSEK supports the LP file format with some extensions. The LP format is not a completely well-defined standard and hence different optimization packages may interpret the same LP file in slightly different ways. **MOSEK** tries to emulate as closely as possible CPLEX's behavior, but tries to stay backward compatible.

The LP file format can specify problems on the form

$$
\begin{array}{lccccc}
\text{minimize/maximize} & & & c^T x + \frac{1}{2} q^o(x) & & \\
\text{subject to} & l^c & \leq & Ax + \frac{1}{2} q(x) & \leq & u^c, \\
 & l^x & \leq & x & \leq & u^x, \\
 & & & x_{\mathcal{J}} \text{ integer}, & &
\end{array}
$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear term in the objective.
- $q^o :\in \mathbb{R}^n \rightarrow \mathbb{R}$ is the quadratic term in the objective where

$$q^o(x) = x^T Q^o x$$

and it is assumed that

$$Q^o = (Q^o)^T.$$

- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $q : \mathbb{R}^n \rightarrow \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = x^T Q^i x$$

where it is assumed that

$$Q^i = (Q^i)^T.$$

- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer constrained variables.

14.1.1 File Sections

An LP formatted file contains a number of sections specifying the objective, constraints, variable bounds, and variable types. The section keywords may be any mix of upper and lower case letters.

Objective Function

The first section beginning with one of the keywords

```
max
maximum
maximize
min
minimum
minimize
```

defines the objective sense and the objective function, i.e.

$$c^T x + \frac{1}{2} x^T Q^o x.$$

The objective may be given a name by writing

```
myname:
```

before the expressions. If no name is given, then the objective is named `obj`.

The objective function contains linear and quadratic terms. The linear terms are written as:

```
4 x1 + x2 - 0.1 x3
```

and so forth. The quadratic terms are written in square brackets (`[ ]`) and are either squared or multiplied as in the examples

```
x1^2
```

and

```
x1 * x2
```

There may be zero or more pairs of brackets containing quadratic expressions.

An example of an objective section is

```
minimize
myobj: 4 x1 + x2 - 0.1 x3 + [ x1^2 + 2.1 x1 * x2 ]/2
```

Please note that the quadratic expressions are multiplied with $\frac{1}{2}$, so that the above expression means

$$\text{minimize} \quad 4x_1 + x_2 - 0.1 \cdot x_3 + \tfrac{1}{2}(x_1^2 + 2.1 \cdot x_1 \cdot x_2)$$

If the same variable occurs more than once in the linear part, the coefficients are added, so that `4 x1 + 2 x1` is equivalent to `6 x1`. In the quadratic expressions `x1 * x2` is equivalent to `x2 * x1` and, as in the linear part, if the same variables multiplied or squared occur several times their coefficients are added.

Constraints

The second section beginning with one of the keywords

```
subj to
subject to
s.t.
st
```

defines the linear constraint matrix A and the quadratic matrices Q^i.

A constraint contains a name (optional), expressions adhering to the same rules as in the objective and a bound:

```
subject to
con1: x1 + x2 + [ x3^2 ]/2 <= 5.1
```

The bound type (here <=) may be any of <, <=, =, >, >= (< and <= mean the same), and the bound may be any number.

In the standard LP format it is not possible to define more than one bound, but **MOSEK** supports defining ranged constraints by using double-colon (`::`) instead of a single-colon (`:`) after the constraint name, i.e.

$$-5 \leq x_1 + x_2 \leq 5 \tag{14.1}$$

may be written as

```
con:: -5 < x_1 + x_2 < 5
```

By default **MOSEK** writes ranged constraints this way.

If the files must adhere to the LP standard, ranged constraints must either be split into upper bounded and lower bounded constraints or be written as an equality with a slack variable. For example the expression (14.1) may be written as

$$x_1 + x_2 - sl_1 = 0, \ -5 \leq sl_1 \leq 5.$$

Bounds

Bounds on the variables can be specified in the bound section beginning with one of the keywords

```
bound
bounds
```

The bounds section is optional but should, if present, follow the `subject to` section. All variables listed in the bounds section must occur in either the objective or a constraint.

The default lower and upper bounds are 0 and $+\infty$. A variable may be declared free with the keyword `free`, which means that the lower bound is $-\infty$ and the upper bound is $+\infty$. Furthermore it may be assigned a finite lower and upper bound. The bound definitions for a given variable may be written in one or two lines, and bounds can be any number or $\pm\infty$ (written as `+inf`/`-inf`/`+infinity`/`-infinity`) as in the example

```
bounds
x1 free
x2 <= 5
0.1 <= x2
x3 = 42
2 <= x4 < +inf
```

Variable Types

The final two sections are optional and must begin with one of the keywords

```
bin
binaries
binary
```

and

```
gen
general
```

Under `general` all integer variables are listed, and under `binary` all binary (integer variables with bounds 0 and 1) are listed:

```
general
x1 x2
binary
x3 x4
```

Again, all variables listed in the binary or general sections must occur in either the objective or a constraint.

Terminating Section

Finally, an LP formatted file must be terminated with the keyword

```
end
```

14.1.2 LP File Examples

Linear example `lo1.lp`

```
\ File: lo1.lp
maximize
obj: 3 x1 + x2 + 5 x3 + x4
subject to
c1:  3 x1 + x2 + 2 x3 = 30
c2:  2 x1 + x2 + 3 x3 + x4 >= 15
c3:  2 x2 + 3 x4 <= 25
bounds
 0 <= x1 <= +infinity
 0 <= x2 <= 10
 0 <= x3 <= +infinity
 0 <= x4 <= +infinity
end
```

Mixed integer example `milo1.lp`

```
maximize
obj: x1 + 6.4e-01 x2
subject to
c1:  5e+01 x1 + 3.1e+01 x2 <= 2.5e+02
c2:  3e+00 x1 - 2e+00 x2 >= -4e+00
bounds
 0 <= x1 <= +infinity
 0 <= x2 <= +infinity
general
 x1 x2
end
```

14.1.3 LP Format peculiarities

Comments

Anything on a line after a \ is ignored and is treated as a comment.

Names

A name for an objective, a constraint or a variable may contain the letters *a-z*, *A-Z*, the digits *0-9* and the characters

```
!"#$%&()/,.;?@_'`|~
```

The first character in a name must not be a number, a period or the letter `e` or `E`. Keywords must not be used as names.

MOSEK accepts any character as valid for names, except `\0`. A name that is not allowed in LP file will be changed and a warning will be issued.

The algorithm for making names LP valid works as follows: The name is interpreted as an `utf-8` string. For a unicode character `c`:

- If `c==_` (underscore), the output is `__` (two underscores).
- If `c` is a valid LP name character, the output is just `c`.
- If `c` is another character in the ASCII range, the output is `_XX`, where `XX` is the hexadecimal code for the character.
- If `c` is a character in the range *127-65535*, the output is `_uXXXX`, where `XXXX` is the hexadecimal code for the character.
- If `c` is a character above 65535, the output is `_UXXXXXXXX`, where `XXXXXXXX` is the hexadecimal code for the character.

Invalid `utf-8` substrings are escaped as `_XX'`, and if a name starts with a period, *e* or *E*, that character is escaped as `_XX`.

Variable Bounds

Specifying several upper or lower bounds on one variable is possible but **MOSEK** uses only the tightest bounds. If a variable is fixed (with `=`), then it is considered the tightest bound.

MOSEK Extensions to the LP Format

Some optimization software packages employ a more strict definition of the LP format than the one used by **MOSEK**. The limitations imposed by the strict LP format are the following:

- Quadratic terms in the constraints are not allowed.
- Names can be only 16 characters long.
- Lines must not exceed 255 characters in length.

To get around some of the inconveniences converting from other problem formats, **MOSEK** allows lines to contain 1024 characters and names may have any length (shorter than the 1024 characters).

14.1.4 Formatting of an LP File

A few parameters control the visual formatting of LP files written by **MOSEK** in order to make it easier to read the files. These parameters are

- *writeLpLineWidth*
- *writeLpTermsPerLine*

The first parameter sets the maximum number of characters on a single line. The default value is 80 corresponding roughly to the width of a standard text document.

The second parameter sets the maximum number of terms per line; a term means a sign, a coefficient, and a name (for example `+ 42 elephants`). The default value is 0, meaning that there is no maximum.

Unnamed Constraints

Reading and writing an LP file with **MOSEK** may change it superficially. If an LP file contains unnamed constraints or objective these are given their generic names when the file is read (however unnamed constraints in **MOSEK** are written without names).

14.2 The MPS File Format

MOSEK supports the standard MPS format with some extensions. For a detailed description of the MPS format see the book by Nazareth *[Naz87]*.

14.2.1 MPS File Structure

The version of the MPS format supported by **MOSEK** allows specification of an optimization problem of the form

$$
\begin{array}{rcccl}
l^c & \leq & Ax + q(x) & \leq & u^c, \\
l^x & \leq & x & \leq & u^x, \\
 & & x \in \mathcal{K}, & & \\
 & & x_{\mathcal{J}} \ \ \text{integer}, & &
\end{array}
\tag{14.2}
$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $q : \mathbb{R}^n \rightarrow \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = \frac{1}{2} x^T Q^i x$$

where it is assumed that

$$Q^i = (Q^i)^T.$$

Please note the explicit $\frac{1}{2}$ in the quadratic term and that Q^i is required to be symmetric.

- $\mathcal{K}$ is a convex cone.
- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer-constrained variables.

An MPS file with one row and one column can be illustrated like this:

```
*         1         2         3         4         5         6
*234567890123456789012345678901234567890123456789012345678901234567890
NAME          [name]
OBJSENSE
[objsense]
OBJNAME
[objname]
ROWS
```

```
?  [cname1]
COLUMNS
[vname1]  [cname1]    [value1]     [vname3]  [value2]
RHS
[name]    [cname1]    [value1]     [cname2]  [value2]
RANGES
[name]    [cname1]    [value1]     [cname2]  [value2]
QSECTION      [cname1]
[vname1]  [vname2]    [value1]     [vname3]  [value2]
QMATRIX
[vname1]  [vname2]    [value1]
QUADOBJ
[vname1]  [vname2]    [value1]
QCMATRIX      [cname1]
[vname1]  [vname2]    [value1]
BOUNDS
?? [name]    [vname1]    [value1]
CSECTION      [kname1]    [value1]     [ktype]
[vname1]
ENDATA
```

Here the names in capitals are keywords of the MPS format and names in brackets are custom defined names or values. A couple of notes on the structure:

- Fields: All items surrounded by brackets appear in *fields*. The fields named "`valueN`" are numerical values. Hence, they must have the format

  ```
    [+|-]XXXXXXX.XXXXXX[[e|E][+|-]XXX]


  where

  .. code-block:: text

     X = [0|1|2|3|4|5|6|7|8|9].
  ```

- Sections: The MPS file consists of several sections where the names in capitals indicate the beginning of a new section. For example, `COLUMNS` denotes the beginning of the columns section.
- Comments: Lines starting with an `*` are comment lines and are ignored by **MOSEK**.
- Keys: The question marks represent keys to be specified later.
- Extensions: The sections `QSECTION` and `CSECTION` are specific **MOSEK** extensions of the MPS format. The sections `QMATRIX`, `QUADOBJ` and `QCMATRIX` are included for sake of compatibility with other vendors extensions to the MPS format.

The standard MPS format is a fixed format, i.e. everything in the MPS file must be within certain fixed positions. **MOSEK** also supports a *free format*. See Section *14.2.9* for details.

Linear example `lo1.mps`

A concrete example of a MPS file is presented below:

```
* File: lo1.mps
NAME          lo1
OBJSENSE
    MAX
ROWS
 N  obj
 E  c1
 G  c2
```

```
 L  c3
COLUMNS
    x1        obj       3
    x1        c1        3
    x1        c2        2
    x2        obj       1
    x2        c1        1
    x2        c2        1
    x2        c3        2
    x3        obj       5
    x3        c1        2
    x3        c2        3
    x4        obj       1
    x4        c2        1
    x4        c3        3
RHS
    rhs       c1        30
    rhs       c2        15
    rhs       c3        25
RANGES
BOUNDS
 UP bound     x2        10
ENDATA
```

Subsequently each individual section in the MPS format is discussed.

Section NAME

In this section a name (`[name]`) is assigned to the problem.

OBJSENSE (optional)

This is an optional section that can be used to specify the sense of the objective function. The `OBJSENSE` section contains one line at most which can be one of the following

```
MIN
MINIMIZE
MAX
MAXIMIZE
```

It should be obvious what the implication is of each of these four lines.

OBJNAME (optional)

This is an optional section that can be used to specify the name of the row that is used as objective function. The `OBJNAME` section contains one line at most which has the form

```
objname
```

`objname` should be a valid row name.

ROWS

A record in the `ROWS` section has the form

```
?  [cname1]
```

where the requirements for the fields are as follows:

Field	Starting Position	Max Width	required	Description
?	2	1	Yes	Constraint key
[cname1]	5	8	Yes	Constraint name

Hence, in this section each constraint is assigned an unique name denoted by `[cname1]`. Please note that `[cname1]` starts in position 5 and the field can be at most 8 characters wide. An initial key `?` must be present to specify the type of the constraint. The key can have the values `E`, `G`, `L`, or `N` with the following interpretation:

Constraint type	l_i^c	u_i^c
E	finite	l_i^c
G	finite	∞
L	$-\infty$	finite
N	$-\infty$	∞

In the MPS format an objective vector is not specified explicitly, but one of the constraints having the key `N` will be used as the objective vector c . In general, if multiple `N` type constraints are specified, then the first will be used as the objective vector c.

COLUMNS

In this section the elements of A are specified using one or more records having the form:

```
[vname1]   [cname1]    [value1]     [cname2]   [value2]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[cname1]	15	8	Yes	Constraint name
[value1]	25	12	Yes	Numerical value
[cname2]	40	8	No	Constraint name
[value2]	50	12	No	Numerical value

Hence, a record specifies one or two elements a_{ij} of A using the principle that `[vname1]` and `[cname1]` determines j and i respectively. Please note that `[cname1]` must be a constraint name specified in the `ROWS` section. Finally, `[value1]` denotes the numerical value of a_{ij} . Another optional element is specified by `[cname2]`, and `[value2]` for the variable specified by `[vname1]`. Some important comments are:

- All elements belonging to one variable must be grouped together.
- Zero elements of A should not be specified.
- At least one element for each variable should be specified.

RHS (optional)

A record in this section has the format

```
[name]     [cname1]    [value1]     [cname2]   [value2]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	required	Description
[name]	5	8	Yes	Name of the RHS vector
[cname1]	15	8	Yes	Constraint name
[value1]	25	12	Yes	Numerical value
[cname2]	40	8	No	Constraint name
[value2]	50	12	No	Numerical value

The interpretation of a record is that `[name]` is the name of the `RHS` vector to be specified. In general, several vectors can be specified. `[cname1]` denotes a constraint name previously specified in the `ROWS` section. Now, assume that this name has been assigned to the i th constraint and v_1 denotes the value specified by `[value1]`, then the interpretation of v_1 is:

Constraint type	l_i^c	u_i^c
E	v_1	v_1
G	v_1	
L		v_1
N		

An optional second element is specified by `[cname2]` and `[value2]` and is interpreted in the same way. Please note that it is not necessary to specify zero elements, because elements are assumed to be zero.

`RANGES` (optional)

A record in this section has the form

```
[name]     [cname1]     [value1]      [cname2]   [value2]
```

where the requirements for each fields are as follows:

Field	Starting Position	Max Width	required	Description
`[name]`	5	8	Yes	Name of the `RANGE` vector
`[cname1]`	15	8	Yes	Constraint name
`[value1]`	25	12	Yes	Numerical value
`[cname2]`	40	8	No	Constraint name
`[value2]`	50	12	No	Numerical value

The records in this section are used to modify the bound vectors for the constraints, i.e. the values in l^c and u^c . A record has the following interpretation: `[name]` is the name of the `RANGE` vector and `[cname1]` is a valid constraint name. Assume that `[cname1]` is assigned to the i th constraint and let v_1 be the value specified by `[value1]`, then a record has the interpretation:

Constraint type	Sign of v_1	l_i^c	u_i^c
E	$-$	$u_i^c + v_1$	
E	$+$		$l_i^c + v_1$
G	$-$ or $+$	$l_i^c + \|v_1\|$	
L	$-$ or $+$	$u_i^c - \|v_1\|$	
N			

`QSECTION` (optional)

Within the `QSECTION` the label `[cname1]` must be a constraint name previously specified in the `ROWS` section. The label `[cname1]` denotes the constraint to which the quadratic term belongs. A record in the `QSECTION` has the form

```
[vname1]   [vname2]     [value1]      [vname3]   [value2]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
`[vname1]`	5	8	Yes	Variable name
`[vname2]`	15	8	Yes	Variable name
`[value1]`	25	12	Yes	Numerical value
`[vname3]`	40	8	No	Variable name
`[value2]`	50	12	No	Numerical value

A record specifies one or two elements in the lower triangular part of the Q^i matrix where [cname1] specifies the i . Hence, if the names [vname1] and [vname2] have been assigned to the k th and j th variable, then Q^i_{kj} is assigned the value given by [value1] An optional second element is specified in the same way by the fields [vname1], [vname3], and [value2].

The example

$$\begin{array}{lccc} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
N  obj
G  c1
COLUMNS
x1        c1        1.0
x2        obj       -1.0
x2        c1        1.0
x3        c1        1.0
RHS
rhs       c1        1.0
QSECTION      obj
x1        x1        2.0
x1        x3        -1.0
x2        x2        0.2
x3        x3        2.0
ENDATA
```

Regarding the QSECTIONs please note that:

- Only one QSECTION is allowed for each constraint.
- The QSECTIONs can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QSECTION must already be specified in the COLUMNS section.
- All entries specified in a QSECTION are assumed to belong to the lower triangular part of the quadratic term of Q .

QMATRIX/QUADOBJ (optional)

The QMATRIX and QUADOBJ sections allow to define the quadratic term of the objective function. They differ in how the quadratic term of the objective function is stored:

- QMATRIX It stores all the nonzeros coefficients, withouot taking advantage of the symmetry of the Q matrix.
- QUADOBJ It only store the upper diagonal nonzero elements of the Q matrix.

A record in both sections has the form:

```
[vname1]  [vname2]    [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[vname2]	15	8	Yes	Variable name
[value1]	25	12	Yes	Numerical value

A record specifies one elements of the Q matrix in the objective function . Hence, if the names [vname1] and [vname2] have been assigned to the k th and j th variable, then Q_{kj} is assigned the value given by [value1]. Note that a line must apper for each off-diagonal coefficient if using a QMATRIX section, while only one entry is required in a QUADOBJ section. The quadratic part of the objective function will be evaluated as $1/2x^T Qx$.

The example

$$\begin{array}{lccl} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation using QMATRIX

```
* File: qo1_matrix.mps
NAME          qo1_qmatrix
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QMATRIX
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

or the following using QUADOBJ

```
* File: qo1_quadobj.mps
NAME          qo1_quadobj
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QUADOBJ
    x1        x1        2.0
    x1        x3        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Please also note that:

- A QMATRIX/QUADOBJ section can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QMATRIX/QUADOBJ section must already be specified in the COLUMNS section.

14.2.2 QCMATRIX (optional)

A QCMATRIX section allows to specify the quadratic part of a given constraints. Within the QCMATRIX the label [cname1] must be a constraint name previously specified in the ROWS section. The label [cname1] denotes the constraint to which the quadratic term belongs. A record in the QSECTION has the form

```
[vname1]  [vname2]    [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[vname2]	15	8	Yes	Variable name
[value1]	25	12	Yes	Numerical value

A record specifies an entry of the Q^i matrix where [cname1] specifies the i. Hence, if the names [vname1] and [vname2] have been assigned to the k th and j th variable, then Q^i_{kj} is assigned the value given by [value1]. Moreover, the quadratic term is represented as $1/2x^TQx$.

The example

$$
\begin{array}{lccc}
\text{minimize} & x_2 & & \\
\text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\
 & \frac{1}{2}(-2x_1x_3 + 0.2x_2^2 + 2x_3^2) & \leq & 10, \\
 & x \geq 0 & &
\end{array}
$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
 L  q1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
    rhs       q1        10.0
QCMATRIX      q1
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Regarding the QCMATRIXs please note that:

- Only one QCMATRIX is allowed for each constraint.
- The QCMATRIXs can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QSECTION must already be specified in the COLUMNS section.
- A QCMATRIX does not exploit the symmetry of Q: an off-diagonal entry (i, j) should apperas twice.

14.2.3 BOUNDS (optional)

In the BOUNDS section changes to the default bounds vectors l^x and u^x are specified. The default bounds vectors are $l^x = 0$ and $u^x = \infty$. Moreover, it is possible to specify several sets of bound vectors. A record in this section has the form

```
?? [name]    [vname1]    [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	Required	Description
??	2	2	Yes	Bound key
[name]	5	8	Yes	Name of the BOUNDS vector
[vname1]	15	8	Yes	Variable name
[value1]	25	12	No	Numerical value

Hence, a record in the BOUNDS section has the following interpretation: [name] is the name of the bound vector and [vname1] is the name of the variable which bounds are modified by the record. ?? and [value1] are used to modify the bound vectors according to the following table:

??	l_j^x	u_j^x	Made integer (added to $\mathcal{J}$)
FR	$-\infty$	∞	No
FX	v_1	v_1	No
LO	v_1	unchanged	No
MI	$-\infty$	unchanged	No
PL	unchanged	∞	No
UP	unchanged	v_1	No
BV	0	1	Yes
LI	$\lceil v_1 \rceil$	unchanged	Yes
UI	unchanged	$\lfloor v_1 \rfloor$	Yes

v_ 1 is the value specified by [value1].

14.2.4 CSECTION (optional)

The purpose of the CSECTION is to specify the constraint

$$x \in \mathcal{K}.$$

in (14.2). It is assumed that $\mathcal{K}$ satisfies the following requirements. Let

$$x^t \in \mathbb{R}^{n^t}, \quad t = 1, \ldots, k$$

be vectors comprised of parts of the decision variables x so that each decision variable is a member of exactly **one** vector x^t, for example

$$x^1 = \begin{bmatrix} x_1 \\ x_4 \\ x_7 \end{bmatrix} \quad \text{and} \quad x^2 = \begin{bmatrix} x_6 \\ x_5 \\ x_3 \\ x_2 \end{bmatrix}.$$

Next define

$$\mathcal{K} := \left\{ x \in \mathbb{R}^n : \quad x^t \in \mathcal{K}_t, \quad t = 1, \ldots, k \right\}$$

where $\mathcal{K}_t$ must have one of the following forms

- $\mathbb{R}$ set:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} \right\}.$$

- Quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1 \geq \sqrt{\sum_{j=2}^{n^t} x_j^2} \right\}. \tag{14.3}$$

- Rotated quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : 2x_1 x_2 \geq \sum_{j=3}^{n^t} x_j^2, \quad x_1, x_2 \geq 0 \right\}. \tag{14.4}$$

In general, only quadratic and rotated quadratic cones are specified in the MPS file whereas membership of the $\mathbb{R}$ set is not. If a variable is not a member of any other cone then it is assumed to be a member of an $\mathbb{R}$ cone.

Next, let us study an example. Assume that the quadratic cone

$$x_4 \geq \sqrt{x_5^2 + x_8^2}$$

and the rotated quadratic cone

$$x_3 x_7 \geq x_1^2 + x_0^2, \quad x_3, x_7 \geq 0,$$

should be specified in the MPS file. One `CSECTION` is required for each cone and they are specified as follows:

```
*         1         2         3         4         5         6
*23456789012345678901234567890123456789012345678901234567890
CSECTION      konea       0.0          QUAD
x4
x5
x8
CSECTION      koneb       0.0          RQUAD
x7
x3
x1
x0
```

This first `CSECTION` specifies the cone (14.3) which is given the name `konea`. This is a quadratic cone which is specified by the keyword `QUAD` in the `CSECTION` header. The 0.0 value in the `CSECTION` header is not used by the `QUAD` cone.

The second `CSECTION` specifies the rotated quadratic cone (14.4). Please note the keyword `RQUAD` in the `CSECTION` which is used to specify that the cone is a rotated quadratic cone instead of a quadratic cone. The 0.0 value in the `CSECTION` header is not used by the `RQUAD` cone.

In general, a `CSECTION` header has the format

```
CSECTION      [kname1]    [value1]     [ktype]
```

where the requirement for each field are as follows:

Field	Starting Position	Max Width	Required	Description
[kname1]	5	8	Yes	Name of the cone
[value1]	15	12	No	Cone parameter
[ktype]	25		Yes	Type of the cone.

The possible cone type keys are:

Cone type key	Members	Interpretation.
QUAD	≤ 1	Quadratic cone i.e. (14.3).
RQUAD	≤ 2	Rotated quadratic cone i.e. (14.4).

Please note that a quadratic cone must have at least one member whereas a rotated quadratic cone must have at least two members. A record in the CSECTION has the format

```
[vname1]
```

where the requirements for each field are

Field	Starting Position	Max Width	required	Description
[vname1]	2	8	Yes	A valid variable name

The most important restriction with respect to the CSECTION is that a variable must occur in only one CSECTION.

14.2.5 ENDATA

This keyword denotes the end of the MPS file.

14.2.6 Integer Variables

Using special bound keys in the BOUNDS section it is possible to specify that some or all of the variables should be integer-constrained i.e. be members of $\mathcal{J}$. However, an alternative method is available.

This method is available only for backward compatibility and we recommend that it is not used. This method requires that markers are placed in the COLUMNS section as in the example:

```
COLUMNS
x1        obj       -10.0          c1        0.7
x1        c2        0.5            c3        1.0
x1        c4        0.1
* Start of integer-constrained variables.
MARK000   'MARKER'                 'INTORG'
x2        obj       -9.0           c1        1.0
x2        c2        0.8333333333   c3        0.66666667
x2        c4        0.25
x3        obj       1.0            c6        2.0
MARK001   'MARKER'                 'INTEND'
```

- End of integer-constrained variables.

Please note that special marker lines are used to indicate the start and the end of the integer variables. Furthermore be aware of the following

- IMPORTANT: All variables between the markers are assigned a default lower bound of 0 and a default upper bound of 1. **This may not be what is intended.** If it is not intended, the correct bounds should be defined in the BOUNDS section of the MPS formatted file.
- **MOSEK** ignores field 1, i.e. MARK0001 and MARK001, however, other optimization systems require them.
- Field 2, i.e. MARKER, must be specified including the single quotes. This implies that no row can be assigned the name MARKER.
- Field 3 is ignored and should be left blank.
- Field 4, i.e. INTORG and INTEND, must be specified.
- It is possible to specify several such integer marker sections within the COLUMNS section.

14.2.7 General Limitations

- An MPS file should be an ASCII file.

14.2.8 Interpretation of the MPS Format

Several issues related to the MPS format are not well-defined by the industry standard. However, **MOSEK** uses the following interpretation:

- If a matrix element in the `COLUMNS` section is specified multiple times, then the multiple entries are added together.
- If a matrix element in a `QSECTION` section is specified multiple times, then the multiple entries are added together.

14.2.9 The Free MPS Format

MOSEK supports a free format variation of the MPS format. The free format is similar to the MPS file format but less restrictive, e.g. it allows longer names. However, it also presents two main limitations:

- A name must not contain any blanks.

14.3 The OPF Format

The *Optimization Problem Format (OPF)* is an alternative to LP and MPS files for specifying optimization problems. It is row-oriented, inspired by the CPLEX LP format.

Apart from containing objective, constraints, bounds etc. it may contain complete or partial solutions, comments and extra information relevant for solving the problem. It is designed to be easily read and modified by hand and to be forward compatible with possible future extensions.

Intended use

The OPF file format is meant to replace several other files:

- The LP file format: Any problem that can be written as an LP file can be written as an OPF file too; furthermore it naturally accommodates ranged constraints and variables as well as arbitrary characters in names, fixed expressions in the objective, empty constraints, and conic constraints.
- Parameter files: It is possible to specify integer, double and string parameters along with the problem (or in a separate OPF file).
- Solution files: It is possible to store a full or a partial solution in an OPF file and later reload it.

14.3.1 The File Format

The format uses tags to structure data. A simple example with the basic sections may look like this:

```
[comment]
This is a comment. You may write almost anything here...
[/comment]

# This is a single-line comment.

[objective min 'myobj']
x + 3 y + x^2 + 3 y^2 + z + 1
[/objective]

[constraints]
[con 'con01'] 4 &<= x + y  [/con]
[/constraints]
```

```
[bounds]
[b] -10 &<= x,y &<= 10  [/b]

[cone quad] x,y,z [/cone]
[/bounds]
```

A scope is opened by a tag of the form `[tag]` and closed by a tag of the form `[/tag]`. An opening tag may accept a list of unnamed and named arguments, for examples:

```
[tag value] tag with one unnamed argument [/tag]
[tag arg=value] tag with one named argument in quotes [/tag]
```

Unnamed arguments are identified by their order, while named arguments may appear in any order, but never before an unnamed argument. The `value` can be a quoted, single-quoted or double-quoted text string, i.e.

```
[tag 'value']     single-quoted value [/tag]
[tag arg='value'] single-quoted value [/tag]
[tag "value"]     double-quoted value [/tag]
[tag arg="value"] double-quoted value [/tag]
```

Sections

The recognized tags are

[comment]

A comment section. This can contain *almost* any text: Between single quotes (') or double quotes (") any text may appear. Outside quotes the markup characters ([and]) must be prefixed by backslashes. Both single and double quotes may appear alone or inside a pair of quotes if it is prefixed by a backslash.

[objective]

The objective function: This accepts one or two parameters, where the first one (in the above example `min`) is either `min` or `max` (regardless of case) and defines the objective sense, and the second one (above `myobj`), if present, is the objective name. The section may contain linear and quadratic expressions. If several objectives are specified, all but the last are ignored.

[constraints]

This does not directly contain any data, but may contain the subsection `con` defining a linear constraint.

`[con]` defines a single constraint; if an argument is present (`[con NAME]`) this is used as the name of the constraint, otherwise it is given a null-name. The section contains a constraint definition written as linear and quadratic expressions with a lower bound, an upper bound, with both or with an equality. Examples:

```
[constraints]
[con 'con1'] 0 <= x + y       [/con]
[con 'con2'] 0 >= x + y       [/con]
[con 'con3'] 0 <= x + y <= 10 [/con]
[con 'con4']      x + y  = 10 [/con]
[/constraints]
```

Constraint names are unique. If a constraint is specified which has the same name as a previously defined constraint, the new constraint replaces the existing one.

[bounds]

This does not directly contain any data, but may contain the subsections `b` (linear bounds on variables) and `cone` (quadratic cone).

`[b]`. Bound definition on one or several variables separated by comma (`,`). An upper or lower bound on a variable replaces any earlier defined bound on that variable. If only one bound (upper or lower) is given only this bound is replaced. This means that upper and lower bounds can be specified separately. So the OPF bound definition:

```
[b]  x,y >= -10  [/b]
[b]  x,y <= 10   [/b]
```

results in the bound $-10 \leq x, y \leq 10$.

`[cone]`. currently supports the *quadratic cone* and the *rotated quadratic cone*.

A conic constraint is defined as a set of variables which belong to a single unique cone.

- A quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^2 > \sum_{i=2}^{n} x_i^2.$$

- A rotated quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1 x_2 > \sum_{i=3}^{n} x_i^2.$$

A `[bounds]`-section example:

```
[bounds]
[b]  0 <= x,y <= 10  [/b] # ranged bound
[b] 10 >= x,y >=  0  [/b] # ranged bound
[b]  0 <= x,y <= inf [/b] # using inf
[b]       x,y free   [/b] # free variables
# Let (x,y,z,w) belong to the cone K
[cone quad]  x,y,z,w  [/cone] # quadratic cone
[cone rquad] x,y,z,w  [/cone] # rotated quadratic cone
[/bounds]
```

By default all variables are free.

[variables]

This defines an ordering of variables as they should appear in the problem. This is simply a space-separated list of variable names.

[integer]

This contains a space-separated list of variables and defines the constraint that the listed variables must be integer values.

[hints]

This may contain only non-essential data; for example estimates of the number of variables, constraints and non-zeros. Placed before all other sections containing data this may reduce the time spent reading the file.

In the `hints` section, any subsection which is not recognized by **MOSEK** is simply ignored. In this section a hint in a subsection is defined as follows:

```
[hint ITEM] value [/hint]
```

where `ITEM` may be replaced by `numvar` (number of variables), `numcon` (number of linear/quadratic constraints), `numanz` (number of linear non-zeros in constraints) and `numqnz` (number of quadratic non-zeros in constraints).

[solutions]

This section can contain a set of full or partial solutions to a problem. Each solution must be specified using a `[solution]`-section, i.e.

```
[solutions]
[solution]...[/solution] #solution 1
[solution]...[/solution] #solution 2
#other solutions....
[solution]...[/solution] #solution n
[/solutions]
```

Note that a `[solution]`-section must be always specified inside a `[solutions]`-section. The syntax of a `[solution]`-section is the following:

```
[solution SOLTYPE status=STATUS]...[/solution]
```

where `SOLTYPE` is one of the strings

- `interior`, a non-basic solution,
- `basic`, a basic solution,
- `integer`, an integer solution,

and `STATUS` is one of the strings

- `UNKNOWN`,
- `OPTIMAL`,
- `INTEGER_OPTIMAL`,
- `PRIM_FEAS`,
- `DUAL_FEAS`,
- `PRIM_AND_DUAL_FEAS`,
- `NEAR_OPTIMAL`,
- `NEAR_PRIM_FEAS`,
- `NEAR_DUAL_FEAS`,
- `NEAR_PRIM_AND_DUAL_FEAS`,
- `PRIM_INFEAS_CER`,
- `DUAL_INFEAS_CER`,
- `NEAR_PRIM_INFEAS_CER`,

- NEAR_DUAL_INFEAS_CER,
- NEAR_INTEGER_OPTIMAL.

Most of these values are irrelevant for input solutions; when constructing a solution for simplex hot-start or an initial solution for a mixed integer problem the safe setting is UNKNOWN.

A [solution]-section contains [con] and [var] sections. Each [con] and [var] section defines solution information for a single variable or constraint, specified as list of KEYWORD/value pairs, in any order, written as

```
KEYWORD=value
```

Allowed keywords are as follows:

- sk. The status of the item, where the value is one of the following strings:
  - LOW, the item is on its lower bound.
  - UPR, the item is on its upper bound.
  - FIX, it is a fixed item.
  - BAS, the item is in the basis.
  - SUPBAS, the item is super basic.
  - UNK, the status is unknown.
  - INF, the item is outside its bounds (infeasible).
- lvl Defines the level of the item.
- sl Defines the level of the dual variable associated with its lower bound.
- su Defines the level of the dual variable associated with its upper bound.
- sn Defines the level of the variable associated with its cone.
- y Defines the level of the corresponding dual variable (for constraints only).

A [var] section should always contain the items sk, lvl, sl and su. Items sl and su are not required for integer solutions.

A [con] section should always contain sk, lvl, sl, su and y.

An example of a solution section

```
[solution basic status=UNKNOWN]
[var x0] sk=LOW    lvl=5.0      [/var]
[var x1] sk=UPR    lvl=10.0     [/var]
[var x2] sk=SUPBAS lvl=2.0  sl=1.5 su=0.0 [/var]

[con c0] sk=LOW    lvl=3.0 y=0.0 [/con]
[con c0] sk=UPR    lvl=0.0 y=5.0 [/con]
[/solution]
```

- [vendor] This contains solver/vendor specific data. It accepts one argument, which is a vendor ID – for **MOSEK** the ID is simply mosek – and the section contains the subsection parameters defining solver parameters. When reading a vendor section, any unknown vendor can be safely ignored. This is described later.

Comments using the # may appear anywhere in the file. Between the # and the following line-break any text may be written, including markup characters.

Numbers

Numbers, when used for parameter values or coefficients, are written in the usual way by the `printf` function. That is, they may be prefixed by a sign (`+` or `-`) and may contain an integer part, decimal part and an exponent. The decimal point is always `.` (a dot). Some examples are

```
1
1.0
.0
1.
1e10
1e+10
1e-10
```

Some *invalid* examples are

```
e10   # invalid, must contain either integer or decimal part
.     # invalid
.e10  # invalid
```

More formally, the following standard regular expression describes numbers as used:

```
[+|-]?([0-9]+[.][0-9]*|[.][0-9]+)([eE][+|-]?[0-9]+)?
```

Names

Variable names, constraint names and objective name may contain arbitrary characters, which in some cases must be enclosed by quotes (single or double) that in turn must be preceded by a backslash. Unquoted names must begin with a letter (`a-z` or `A-Z`) and contain only the following characters: the letters `a-z` and `A-Z`, the digits `0-9`, braces (`{` and `}`) and underscore (`_`).

Some examples of legal names:

```
an_unquoted_name
another_name{123}
'single quoted name'
"double quoted name"
"name with \\"quote\\" in it"
"name with []s in it"
```

14.3.2 Parameters Section

In the `vendor` section solver parameters are defined inside the `parameters` subsection. Each parameter is written as

```
[p PARAMETER_NAME] value [/p]
```

where `PARAMETER_NAME` is replaced by a **MOSEK** parameter name, usually of the form `MSK_IPAR_...`, `MSK_DPAR_...` or `MSK_SPAR_...`, and the `value` is replaced by the value of that parameter; both integer values and named values may be used. Some simple examples are

```
[vendor mosek]
[parameters]
[p MSK_IPAR_OPF_MAX_TERMS_PER_LINE] 10     [/p]
[p MSK_IPAR_OPF_WRITE_PARAMETERS]   MSK_ON [/p]
[p MSK_DPAR_DATA_TOL_BOUND_INF]     1.0e18 [/p]
[/parameters]
[/vendor]
```

14.3.3 Writing OPF Files from MOSEK

To write an OPF file add the `.opf` extension to the file name.

14.3.4 Examples

This section contains a set of small examples written in OPF and describing how to formulate linear, quadratic and conic problems.

Linear Example `lo1.opf`

Consider the example:

$$\begin{array}{lccccccccc} \text{maximize} & 3x_0 & + & 1x_1 & + & 5x_2 & + & 1x_3 & & \\ \text{subject to} & 3x_0 & + & 1x_1 & + & 2x_2 & & & = & 30, \\ & 2x_0 & + & 1x_1 & + & 3x_2 & + & 1x_3 & \geq & 15, \\ & & & 2x_1 & & & + & 3x_3 & \leq & 25, \end{array}$$

having the bounds

$$\begin{array}{ccccc} 0 & \leq & x_0 & \leq & \infty, \\ 0 & \leq & x_1 & \leq & 10, \\ 0 & \leq & x_2 & \leq & \infty, \\ 0 & \leq & x_3 & \leq & \infty. \end{array}$$

In the `OPF` format the example is displayed as shown in Listing 14.1.

Listing 14.1: Example of an OPF file for a linear problem.

```
[comment]
  The lo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 4 [/hint]
  [hint NUMCON] 3 [/hint]
  [hint NUMANZ] 9 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3 x4
[/variables]

[objective maximize 'obj']
   3 x1 + x2 + 5 x3 + x4
[/objective]

[constraints]
  [con 'c1'] 3 x1 +   x2 + 2 x3         = 30 [/con]
  [con 'c2'] 2 x1 +   x2 + 3 x3 +   x4 >= 15 [/con]
  [con 'c3']        2 x2        + 3 x4 <= 25 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
  [b] 0 <= x2 <= 10 [/b]
[/bounds]
```

Quadratic Example qo1.opf

An example of a quadratic optimization problem is

$$\begin{array}{lccl}\text{minimize} & & & x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\ \text{subject to} & 1 & \leq & x_1 + x_2 + x_3, \\ & & & x \geq 0.\end{array}$$

This can be formulated in `opf` as shown below.

Listing 14.2: Example of an OPF file for a quadratic problem.

```
[comment]
  The qo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 3 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
  [hint NUMQNZ] 4 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3
[/variables]

[objective minimize 'obj']
  # The quadratic terms are often written with a factor of 1/2 as here,
  # but this is not required.

   - x2 + 0.5 ( 2.0 x1 ^ 2 - 2.0 x3 * x1 + 0.2 x2 ^ 2 + 2.0 x3 ^ 2 )
[/objective]

[constraints]
  [con 'c1'] 1.0 <= x1 + x2 + x3 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]
```

Conic Quadratic Example cqo1.opf

Consider the example:

$$\begin{array}{lccc}\text{minimize} & x_3 + x_4 + x_5 & & \\ \text{subject to} & x_0 + x_1 + 2x_2 & = & 1, \\ & x_0, x_1, x_2 & \geq & 0, \\ & x_3 \geq \sqrt{x_0^2 + x_1^2}, & & \\ & 2x_4x_5 \geq x_2^2. & & \end{array}$$

Please note that the type of the cones is defined by the parameter to `[cone ...]`; the content of the `cone`-section is the names of variables that belong to the cone. The resulting OPF file is in Listing 14.3.

Listing 14.3: Example of an OPF file for a conic quadratic problem.

```
[comment]
  The cqo1 example in OPF format.
[/comment]

[hints]
```

```
  [hint NUMVAR] 6 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3 x4 x5 x6
[/variables]

[objective minimize 'obj']
   x4 + x5 + x6
[/objective]

[constraints]
  [con 'c1']  x1 + x2 + 2e+00 x3 = 1e+00 [/con]
[/constraints]

[bounds]
  # We let all variables default to the positive orthant
  [b] 0 <= * [/b]

  # ...and change those that differ from the default
  [b] x4,x5,x6 free [/b]

  # Define quadratic cone: x4 >= sqrt( x1^2 + x2^2 )
  [cone quad 'k1'] x4, x1, x2 [/cone]

  # Define rotated quadratic cone: 2 x5 x6 >= x3^2
  [cone rquad 'k2'] x5, x6, x3 [/cone]
[/bounds]
```

Mixed Integer Example `milo1.opf`

Consider the mixed integer problem:

$$
\begin{array}{lccl}
\text{maximize} & x_0 + 0.64x_1 & & \\
\text{subject to} & 50x_0 + 31x_1 & \leq & 250, \\
 & 3x_0 - 2x_1 & \geq & -4, \\
 & x_0, x_1 \geq 0 & & \text{and integer}
\end{array}
$$

This can be implemented in `OPF` with the file in Listing 14.4.

Listing 14.4: Example of an OPF file for a mixed-integer linear problem.

```
[comment]
  The milo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 2 [/hint]
  [hint NUMCON] 2 [/hint]
  [hint NUMANZ] 4 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2
[/variables]

[objective maximize 'obj']
   x1 + 6.4e-1 x2
[/objective]
```

```
[constraints]
  [con 'c1'] 5e+1 x1 + 3.1e+1 x2 <= 2.5e+2 [/con]
  [con 'c2'] -4 <= 3 x1 - 2 x2 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]

[integer]
  x1 x2
[/integer]
```

14.4 The CBF Format

This document constitutes the technical reference manual of the *Conic Benchmark Format* with file extension: `.cbf` or `.CBF`. It unifies linear, second-order cone (also known as conic quadratic) and semidefinite optimization with mixed-integer variables. The format has been designed with benchmark libraries in mind, and therefore focuses on compact and easily parsable representations. The problem structure is separated from the problem data, and the format moreover facilitates benchmarking of hotstart capability through sequences of changes.

14.4.1 How Instances Are Specified

This section defines the spectrum of conic optimization problems that can be formulated in terms of the keywords of the CBF format.

In the CBF format, conic optimization problems are considered in the following form:

$$\begin{array}{lll} \min / \max & g^{obj} & \\ & g_i \in \mathcal{K}_i, & i \in \mathcal{I}, \\ \text{s.t.} & G_i \in \mathcal{K}_i, & i \in \mathcal{I}^{PSD}, \\ & x_j \in \mathcal{K}_j, & j \in \mathcal{J}, \\ & \overline{X}_j \in \mathcal{K}_j, & j \in \mathcal{J}^{PSD}. \end{array} \tag{14.5}$$

- **Variables** are either scalar variables, x_j for $j \in \mathcal{J}$, or variables, $\overline{X}_j$ for $j \in \mathcal{J}^{PSD}$. Scalar variables can also be declared as integer.
- **Constraints** are affine expressions of the variables, either scalar-valued g_i for $i \in \mathcal{I}$, or matrix-valued G_i for $i \in \mathcal{I}^{PSD}$

$$g_i = \sum_{j \in \mathcal{J}^{PSD}} \langle F_{ij}, X_j \rangle + \sum_{j \in \mathcal{J}} a_{ij} x_j + b_i,$$

$$G_i = \sum_{j \in \mathcal{J}} x_j H_{ij} + D_i.$$

- The **objective function** is a scalar-valued affine expression of the variables, either to be minimized or maximized. We refer to this expression as g^{obj}

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \langle F_j^{obj}, X_j \rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj}.$$

CBF format can represent the following cones $\mathcal{K}$:

- **Free domain** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n\}, \text{ for } n \geq 1.$$

- **Positive orthant** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j \geq 0 \text{ for } j = 1, \dots, n\}, \text{ for } n \geq 1.$$

- **Negative orthant** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j \leq 0 \text{ for } j = 1, \dots, n\}, \text{ for } n \geq 1.$$

- **Fixpoint zero** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j = 0 \text{ for } j = 1, \dots, n\}, \text{ for } n \geq 1.$$

- **Quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \left(\begin{array}{c} p \\ x \end{array} \right) \in \mathbb{R} \times \mathbb{R}^{n-1},\ p^2 \geq x^T x,\ p \geq 0 \right\}, \text{ for } n \geq 2.$$

- **Rotated quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \left(\begin{array}{c} p \\ q \\ x \end{array} \right) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n-2},\ 2pq \geq x^T x,\ p \geq 0,\ q \geq 0 \right\}, \text{ for } n \geq 3.$$

14.4.2 The Structure of CBF Files

This section defines how information is written in the CBF format, without being specific about the type of information being communicated.

All information items belong to exactly one of the three groups of information. These information groups, and the order they must appear in, are:

1. File format.
2. Problem structure.
3. Problem data.

The first group, file format, provides information on how to interpret the file. The second group, problem structure, provides the information needed to deduce the type and size of the problem instance. Finally, the third group, problem data, specifies the coefficients and constants of the problem instance.

Information items

The format is composed as a list of information items. The first line of an information item is the `KEYWORD`, revealing the type of information provided. The second line - of some keywords only - is the `HEADER`, typically revealing the size of information that follows. The remaining lines are the `BODY` holding the actual information to be specified.

```
KEYWORD
BODY

KEYWORD
HEADER
BODY
```

The KEYWORD determines how each line in the HEADER and BODY is structured. Moreover, the number of lines in the BODY follows either from the KEYWORD, the HEADER, or from another information item required to precede it.

Embedded hotstart-sequences

A sequence of problem instances, based on the same problem structure, is within a single file. This is facilitated via the CHANGE within the problem data information group, as a separator between the information items of each instance. The information items following a CHANGE keyword are appending to, or changing (e.g., setting coefficients back to their default value of zero), the problem data of the preceding instance.

The sequence is intended for benchmarking of hotstart capability, where the solvers can reuse their internal state and solution (subject to the achieved accuracy) as warmpoint for the succeeding instance. Whenever this feature is unsupported or undesired, the keyword CHANGE should be interpreted as the end of file.

File encoding and line width restrictions

The format is based on the US-ASCII printable character set with two extensions as listed below. Note, by definition, that none of these extensions can be misinterpreted as printable US-ASCII characters:

- A line feed marks the end of a line, carriage returns are ignored.
- Comment-lines may contain unicode characters in UTF-8 encoding.

The line width is restricted to 512 bytes, with 3 bytes reserved for the potential carriage return, line feed and null-terminator.

Integers and floating point numbers must follow the ISO C decimal string representation in the standard C locale. The format does not impose restrictions on the magnitude of, or number of significant digits in numeric data, but the use of 64-bit integers and 64-bit IEEE 754 floating point numbers should be sufficient to avoid loss of precision.

Comment-line and whitespace rules

The format allows single-line comments respecting the following rule:

- Lines having first byte equal to '#' (US-ASCII 35) are comments, and should be ignored. Comments are only allowed between information items.

Given that a line is not a comment-line, whitespace characters should be handled according to the following rules:

- Leading and trailing whitespace characters should be ignored.
  - The seperator between multiple pieces of information on one line, is either one or more whitespace characters.
- Lines containing only whitespace characters are empty, and should be ignored. Empty lines are only allowed between information items.

14.4.3 Problem Specification

The problem structure

The problem structure defines the objective sense, whether it is minimization and maximization. It also defines the index sets, $\mathcal{J}$, $\mathcal{J}^{PSD}$, $\mathcal{I}$ and $\mathcal{I}^{PSD}$, which are all numbered from zero, $\{0, 1, \ldots\}$, and empty until explicitly constructed.

- **Scalar variables** are constructed in vectors restricted to a conic domain, such as $(x_0, x_1) \in \mathbb{R}^2_+$, $(x_2, x_3, x_4) \in \mathcal{Q}^3$, etc. In terms of the Cartesian product, this generalizes to

$$x \in \mathcal{K}_1^{n_1} \times \mathcal{K}_2^{n_2} \times \cdots \times \mathcal{K}_k^{n_k}$$

  which in the CBF format becomes:

```
VAR
n k
K1 n1
K2 n2
...
Kk nk
```

  where $\sum_i n_i = n$ is the total number of scalar variables. The list of supported cones is found in Table 14.3. Integrality of scalar variables can be specified afterwards.

- **PSD variables** are constructed one-by-one. That is, $X_j \succeq \mathbf{0}^{n_j \times n_j}$ for $j \in \mathcal{J}^{PSD}$, constructs a matrix-valued variable of size $n_j \times n_j$ restricted to be symmetric positive semidefinite. In the CBF format, this list of constructions becomes:

```
PSDVAR
N
n1
n2
...
nN
```

  where N is the total number of PSD variables.

- **Scalar constraints** are constructed in vectors restricted to a conic domain, such as $(g_0, g_1) \in \mathbb{R}^2_+$, $(g_2, g_3, g_4) \in \mathcal{Q}^3$, etc. In terms of the Cartesian product, this generalizes to

$$g \in \mathcal{K}_1^{m_1} \times \mathcal{K}_2^{m_2} \times \cdots \times \mathcal{K}_k^{m_k}$$

  which in the CBF format becomes:

```
CON
m k
K1 m1
K2 m2
..
Kk mk
```

  where $\sum_i m_i = m$ is the total number of scalar constraints. The list of supported cones is found in Table 14.3.

- **PSD constraints** are constructed one-by-one. That is, $G_i \succeq \mathbf{0}^{m_i \times m_i}$ for $i \in \mathcal{I}^{PSD}$, constructs a matrix-valued affine expressions of size $m_i \times m_i$ restricted to be symmetric positive semidefinite. In the CBF format, this list of constructions becomes

```
PSDCON
M
m1
```

```
m2
..
mM
```

where M is the total number of PSD constraints.

With the objective sense, variables (with integer indications) and constraints, the definitions of the many affine expressions follow in problem data.

Problem data

The problem data defines the coefficients and constants of the affine expressions of the problem instance. These are considered zero until explicitly defined, implying that instances with no keywords from this information group are, in fact, valid. Duplicating or conflicting information is a failure to comply with the standard. Consequently, two coefficients written to the same position in a matrix (or to transposed positions in a symmetric matrix) is an error.

The affine expressions of the objective, g^{obj}, of the scalar constraints, g_i, and of the PSD constraints, G_i, are defined separately. The following notation uses the standard trace inner product for matrices, $\langle X, Y \rangle = \sum_{i,j} X_{ij} Y_{ij}$.

- The affine expression of the objective is defined as

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \langle F_j^{obj}, X_j \rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj},$$

  in terms of the symmetric matrices, F_j^{obj}, and scalars, a_j^{obj} and b^{obj}.

- The affine expressions of the scalar constraints are defined, for $i \in \mathcal{I}$, as

$$g_i = \sum_{j \in \mathcal{J}^{PSD}} \langle F_{ij}, X_j \rangle + \sum_{j \in \mathcal{J}} a_{ij} x_j + b_i,$$

  in terms of the symmetric matrices, F_{ij}, and scalars, a_{ij} and b_i.

- The affine expressions of the PSD constraints are defined, for $i \in \mathcal{I}^{PSD}$, as

$$G_i = \sum_{j \in \mathcal{J}} x_j H_{ij} + D_i,$$

  in terms of the symmetric matrices, H_{ij} and D_i.

List of cones

The format uses an explicit syntax for symmetric positive semidefinite cones as shown above. For scalar variables and constraints, constructed in vectors, the supported conic domains and their minimum sizes are given as follows.

Table 14.3: Cones available in the CBF format

Name	CBF keyword	Cone family
Free domain	`F`	linear
Positive orthant	`L+`	linear
Negative orthant	`L-`	linear
Fixpoint zero	`L=`	linear
Quadratic cone	`Q`	second-order
Rotated quadratic cone	`QR`	second-order

14.4.4 File Format Keywords

VER

Description: The version of the Conic Benchmark Format used to write the file.

`HEADER`: None

`BODY`: One line formatted as:

```
INT
```

This is the version number.

Must appear exactly once in a file, as the first keyword.

OBJSENSE

Description: Define the objective sense.

`HEADER`: None

`BODY`: One line formatted as:

```
STR
```

having `MIN` indicates minimize, and `MAX` indicates maximize. Capital letters are required.

Must appear exactly once in a file.

PSDVAR

Description: Construct the PSD variables.

`HEADER`: One line formatted as:

```
INT
```

This is the number of PSD variables in the problem.

`BODY`: A list of lines formatted as:

```
INT
```

This indicates the number of rows (equal to the number of columns) in the matrix-valued PSD variable. The number of lines should match the number stated in the header.

VAR

Description: Construct the scalar variables.

`HEADER`: One line formatted as:

```
INT INT
```

This is the number of scalar variables, followed by the number of conic domains they are restricted to.

`BODY`: A list of lines formatted as:

```
STR INT
```

This indicates the cone name (see Table 14.3), and the number of scalar variables restricted to this cone. These numbers should add up to the number of scalar variables stated first in the header. The number of lines should match the second number stated in the header.

INT

Description: Declare integer requirements on a selected subset of scalar variables.

`HEADER`: one line formatted as:

```
INT
```

This is the number of integer scalar variables in the problem.

`BODY`: a list of lines formatted as:

```
INT
```

This indicates the scalar variable index $j \in \mathcal{J}$. The number of lines should match the number stated in the header.

Can only be used after the keyword `VAR`.

PSDCON

Description: Construct the PSD constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of PSD constraints in the problem.

`BODY`: A list of lines formatted as:

```
INT
```

This indicates the number of rows (equal to the number of columns) in the matrix-valued affine expression of the PSD constraint. The number of lines should match the number stated in the header.

Can only be used after these keywords: `PSDVAR`, `VAR`.

CON

Description: Construct the scalar constraints.

`HEADER`: One line formatted as:

```
INT INT
```

This is the number of scalar constraints, followed by the number of conic domains they restrict to.

`BODY`: A list of lines formatted as:

```
STR INT
```

This indicates the cone name (see Table 14.3), and the number of affine expressions restricted to this cone. These numbers should add up to the number of scalar constraints stated first in the header. The number of lines should match the second number stated in the header.

Can only be used after these keywords: `PSDVAR`, `VAR`

OBJFCOORD

Description: Input sparse coordinates (quadruplets) to define the symmetric matrices F_j^{obj}, as used in the objective.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT INT INT REAL
```

This indicates the PSD variable index $j \in \mathcal{J}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

OBJACOORD

Description: Input sparse coordinates (pairs) to define the scalars, a_j^{obj}, as used in the objective.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT REAL
```

This indicates the scalar variable index $j \in \mathcal{J}$ and the coefficient value. The number of lines should match the number stated in the header.

OBJBCOORD

Description: Input the scalar, b^{obj}, as used in the objective.

`HEADER`: None.

`BODY`: One line formatted as:

```
REAL
```

This indicates the coefficient value.

FCOORD

Description: Input sparse coordinates (quintuplets) to define the symmetric matrices, F_{ij}, as used in the scalar constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT INT INT INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$, the PSD variable index $j \in \mathcal{J}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

ACOORD

Description: Input sparse coordinates (triplets) to define the scalars, a_{ij}, as used in the scalar constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$, the scalar variable index $j \in \mathcal{J}$ and the coefficient value. The number of lines should match the number stated in the header.

BCOORD

Description: Input sparse coordinates (pairs) to define the scalars, b_i, as used in the scalar constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$ and the coefficient value. The number of lines should match the number stated in the header.

HCOORD

Description: Input sparse coordinates (quintuplets) to define the symmetric matrices, H_{ij}, as used in the PSD constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as

```
INT INT INT INT REAL
```

This indicates the PSD constraint index $i \in \mathcal{I}^{PSD}$, the scalar variable index $j \in \mathcal{J}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

DCOORD

Description: Input sparse coordinates (quadruplets) to define the symmetric matrices, D_i, as used in the PSD constraints.

HEADER: One line formatted as

```
INT
```

This is the number of coordinates to be specified.

BODY: A list of lines formatted as:

```
INT INT INT REAL
```

This indicates the PSD constraint index $i \in \mathcal{I}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

CHANGE

Start of a new instance specification based on changes to the previous. Can be interpreted as the end of file when the hotstart-sequence is unsupported or undesired.

BODY: None

Header: None

14.4.5 CBF Format Examples

Minimal Working Example

The conic optimization problem (14.6) , has three variables in a quadratic cone - first one is integer - and an affine expression in domain 0 (equality constraint).

$$\begin{array}{ll} \text{minimize} & 5.1\,x_0 \\ \text{subject to} & 6.2\,x_1 + 7.3\,x_2 - 8.4 \in \{0\} \\ & x \in \mathcal{Q}^3,\ x_0 \in \mathbb{Z}. \end{array} \tag{14.6}$$

Its formulation in the Conic Benchmark Format begins with the version of the CBF format used, to safeguard against later revisions.

```
VER
1
```

Next follows the problem structure, consisting of the objective sense, the number and domain of variables, the indices of integer variables, and the number and domain of scalar-valued affine expressions (i.e., the equality constraint).

```
OBJSENSE
MIN

VAR
3 1
Q 3

INT
1
0

CON
1 1
L= 1
```

Finally follows the problem data, consisting of the coefficients of the objective, the coefficients of the constraints, and the constant terms of the constraints. All data is specified on a sparse coordinate form.

```
OBJACOORD
1
0 5.1

ACOORD
2
0 1 6.2
0 2 7.3

BCOORD
1
0 -8.4
```

This concludes the example.

Mixing Linear, Second-order and Semidefinite Cones

The conic optimization problem (14.7), has a semidefinite cone, a quadratic cone over unordered subindices, and two equality constraints.

$$
\begin{array}{lrcl}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, X_1 \right\rangle + x_1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 & = & 1.0\,, \\
& \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, X_1 \right\rangle + x_0 + x_2 & = & 0.5\,, \\
& x_1 \geq \sqrt{x_0^2 + x_2^2}\,, & & \\
& X_1 \succeq \mathbf{0}\,. & &
\end{array}
\tag{14.7}
$$

The equality constraints are easily rewritten to the conic form, $(g_0, g_1) \in \{0\}^2$, by moving constants such that the right-hand-side becomes zero. The quadratic cone does not fit under the `VAR` keyword in this variable permutation. Instead, it takes a scalar constraint $(g_2, g_3, g_4) = (x_1, x_0, x_2) \in \mathcal{Q}^3$, with scalar variables constructed as $(x_0, x_1, x_2) \in \mathbb{R}^3$. Its formulation in the CBF format is reported in the following list

```
# File written using this version of the Conic Benchmark Format:
#     | Version 1.
VER
1

# The sense of the objective is:
#     | Minimize.
OBJSENSE
MIN

# One PSD variable of this size:
#     | Three times three.
PSDVAR
1
3

# Three scalar variables in this one conic domain:
#     | Three are free.
```

```
VAR
3 1
F 3

# Five scalar constraints with affine expressions in two conic domains:
#     | Two are fixed to zero.
#     | Three are in conic quadratic domain.
CON
5 2
L= 2
Q 3

# Five coordinates in F^{obj}_j coefficients:
#     | F^{obj}[0][0,0] = 2.0
#     | F^{obj}[0][1,0] = 1.0
#     | and more...
OBJFCOORD
5
0 0 0 2.0
0 1 0 1.0
0 1 1 2.0
0 2 1 1.0
0 2 2 2.0

# One coordinate in a^{obj}_j coefficients:
#     | a^{obj}[1] = 1.0
OBJACOORD
1
1 1.0

# Nine coordinates in F_ij coefficients:
#     | F[0,0][0,0] = 1.0
#     | F[0,0][1,1] = 1.0
#     | and more...
FCOORD
9
0 0 0 0 1.0
0 0 1 1 1.0
0 0 2 2 1.0
1 0 0 0 1.0
1 0 1 0 1.0
1 0 2 0 1.0
1 0 1 1 1.0
1 0 2 1 1.0
1 0 2 2 1.0

# Six coordinates in a_ij coefficients:
#     | a[0,1] = 1.0
#     | a[1,0] = 1.0
#     | and more...
ACOORD
6
0 1 1.0
1 0 1.0
1 2 1.0
2 1 1.0
3 0 1.0
4 2 1.0

# Two coordinates in b_i coefficients:
#     | b[0] = -1.0
#     | b[1] = -0.5
BCOORD
```

```
2
0 -1.0
1 -0.5
```

Mixing Semidefinite Variables and Linear Matrix Inequalities

The standard forms in semidefinite optimization are usually based either on semidefinite variables or linear matrix inequalities. In the CBF format, both forms are supported and can even be mixed as shown in.

$$
\begin{array}{lll}
\text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 + x_2 + 1 & \\
\text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X_1 \right\rangle - x_1 - x_2 & \geq \ 0.0\,, \\
 & x_1 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq \ \mathbf{0}\,, \\
 & X_1 \succeq \mathbf{0}\,. &
\end{array}
\tag{14.8}
$$

Its formulation in the CBF format is written in what follows

```
# File written using this version of the Conic Benchmark Format:
#     | Version 1.
VER
1

# The sense of the objective is:
#     | Minimize.
OBJSENSE
MIN

# One PSD variable of this size:
#     | Two times two.
PSDVAR
1
2

# Two scalar variables in this one conic domain:
#     | Two are free.
VAR
2 1
F 2

# One PSD constraint of this size:
#     | Two times two.
PSDCON
1
2

# One scalar constraint with an affine expression in this one conic domain:
#     | One is greater than or equal to zero.
CON
1 1
L+ 1

# Two coordinates in F^{obj}_j coefficients:
#     | F^{obj}[0][0,0] = 1.0
#     | F^{obj}[0][1,1] = 1.0
OBJFCOORD
2
0 0 0 1.0
0 1 1 1.0
```

```
# Two coordinates in a^{obj}_j coefficients:
#      | a^{obj}[0] = 1.0
#      | a^{obj}[1] = 1.0
OBJACOORD
2
0 1.0
1 1.0

# One coordinate in b^{obj} coefficient:
#      | b^{obj} = 1.0
OBJBCOORD
1.0

# One coordinate in F_ij coefficients:
#      | F[0,0][1,0] = 1.0
FCOORD
1
0 0 1 0 1.0

# Two coordinates in a_ij coefficients:
#      | a[0,0] = -1.0
#      | a[0,1] = -1.0
ACOORD
2
0 0 -1.0
0 1 -1.0

# Four coordinates in H_ij coefficients:
#      | H[0,0][1,0] = 1.0
#      | H[0,0][1,1] = 3.0
#      | and more...
HCOORD
4
0 0 1 0 1.0
0 0 1 1 3.0
0 1 0 0 3.0
0 1 1 0 1.0

# Two coordinates in D_i coefficients:
#      | D[0][0,0] = -1.0
#      | D[0][1,1] = -1.0
DCOORD
2
0 0 0 -1.0
0 1 1 -1.0
```

Optimization Over a Sequence of Objectives

The linear optimization problem (14.9), is defined for a sequence of objectives such that hotstarting from one to the next might be advantages.

$$\begin{array}{lccc} \text{maximize}_k & g_k^{obj} & & \\ \text{subject to} & 50\,x_0 + 31 & \leq & 250\,, \\ & 3\,x_0 - 2x_1 & \geq & -4\,, \\ & x \in \mathbb{R}_+^2, & & \end{array} \tag{14.9}$$

given,

1. $g_0^{obj} = x_0 + 0.64x_1$.
2. $g_1^{obj} = 1.11x_0 + 0.76x_1$.

3. $g_2^{obj} = 1.11x_0 + 0.85x_1$.

Its formulation in the CBF format is reported in Listing 14.5.

Listing 14.5: Problem (14.9) in CBF format.

```
# File written using this version of the Conic Benchmark Format:
#     | Version 1.
VER
1

# The sense of the objective is:
#     | Maximize.
OBJSENSE
MAX

# Two scalar variables in this one conic domain:
#     | Two are nonnegative.
VAR
2 1
L+ 2

# Two scalar constraints with affine expressions in these two conic domains:
#     | One is in the nonpositive domain.
#     | One is in the nonnegative domain.
CON
2 2
L- 1
L+ 1

# Two coordinates in a^{obj}_j coefficients:
#     | a^{obj}[0] = 1.0
#     | a^{obj}[1] = 0.64
OBJACOORD
2
0 1.0
1 0.64

# Four coordinates in a_ij coefficients:
#     | a[0,0] = 50.0
#     | a[1,0] = 3.0
#     | and more...
ACOORD
4
0 0 50.0
1 0 3.0
0 1 31.0
1 1 -2.0

# Two coordinates in b_i coefficients:
#     | b[0] = -250.0
#     | b[1] = 4.0
BCOORD
2
0 -250.0
1 4.0

# New problem instance defined in terms of changes.
CHANGE

# Two coordinate changes in a^{obj}_j coefficients. Now it is:
#     | a^{obj}[0] = 1.11
#     | a^{obj}[1] = 0.76
OBJACOORD
```

```
2
0 1.11
1 0.76

# New problem instance defined in terms of changes.
CHANGE

# One coordinate change in a^{obj}_j coefficients. Now it is:
#     | a^{obj}[0] = 1.11
#     | a^{obj}[1] = 0.85
OBJACOORD
1
1 0.85
```

14.5 The XML (OSiL) Format

MOSEK can write data in the standard OSiL xml format. For a definition of the OSiL format please see http://www.optimizationservices.org/.

Only linear constraints (possibly with integer variables) are supported. By default output files with the extension `.xml` are written in the OSiL format.

14.6 The Task Format

The Task format is **MOSEK**'s native binary format. It contains a complete image of a **MOSEK** task, i.e.

- Problem data: Linear, conic quadratic, semidefinite and quadratic data
- Problem item names: Variable names, constraints names, cone names etc.
- Parameter settings
- Solutions

There are a few things to be aware of:

- The task format *does not* support General Convex problems since these are defined by arbitrary user-defined functions.
- Status of a solution read from a file will *always* be unknown.

The format is based on the *TAR* (USTar) file format. This means that the individual pieces of data in a `.task` file can be examined by unpacking it as a *TAR* file. Please note that the inverse may not work: Creating a file using *TAR* will most probably not create a valid **MOSEK** Task file since the order of the entries is important.

14.7 The JSON Format

MOSEK provides the possibility to read/write problems in valid JSON format.

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

The official JSON website http://www.json.org provides plenty of information along with the format definition.

MOSEK defines two JSON-like formats:

- *jtask*
- *jsol*

Warning: Despite being text-based human-readable formats, *jtask* and *jsol* files will include no indentation and no new-lines, in order to keep the files as compact as possible. We therefore strongly advise to use JSON viewer tools to inspect *jtask* and *jsol* files.

14.7.1 *jtask* format

It stores a problem instance. The *jtask* format contains the same information as a *task format*.

Even though a *jtask* file is human-readable, we do not recommend users to create it by hand, but to rely on **MOSEK**.

14.7.2 *jsol* format

It stores a problem solution. The *jsol* format contains all solutions and information items.

14.7.3 A *jtask* example

In Listing 14.6 we present a file in the *jtask* format that corresponds to the sample problem from `lo1.lp`. The listing has been formatted for readability.

Listing 14.6: A formatted *jtask* file for the `lo1.lp` example.

```
{
    "$schema":"http://mosek.com/json/schema#",
    "Task/INFO":{
        "taskname":"lo1",
        "numvar":4,
        "numcon":3,
        "numcone":0,
        "numbarvar":0,
        "numanz":9,
        "numsymmat":0,
        "mosekver":[
            8,
            0,
            0,
            9
        ]
    },
    "Task/data":{
        "var":{
            "name":[
                "x1",
                "x2",
                "x3",
                "x4"
            ],
            "bk":[
                "lo",
                "ra",
```

```
                "lo",
                "lo"
            ],
            "bl":[
                0.0,
                0.0,
                0.0,
                0.0
            ],
            "bu":[
                1e+30,
                1e+1,
                1e+30,
                1e+30
            ],
            "type":[
                "cont",
                "cont",
                "cont",
                "cont"
            ]
        },
        "con":{
            "name":[
                "c1",
                "c2",
                "c3"
            ],
            "bk":[
                "fx",
                "lo",
                "up"
            ],
            "bl":[
                3e+1,
                1.5e+1,
                    -1e+30
            ],
            "bu":[
                3e+1,
                1e+30,
                2.5e+1
            ]
        },
        "objective":{
            "sense":"max",
            "name":"obj",
            "c":{
                "subj":[
                    0,
                    1,
                    2,
                    3
                ],
                "val":[
                    3e+0,
                    1e+0,
                    5e+0,
                    1e+0
                ]
            },
            "cfix":0.0
        },
```

```
        "A":{
            "subi":[
                0,
                0,
                0,
                1,
                1,
                1,
                1,
                2,
                2
            ],
            "subj":[
                0,
                1,
                2,
                0,
                1,
                2,
                3,
                1,
                3
            ],
            "val":[
                3e+0,
                1e+0,
                2e+0,
                2e+0,
                1e+0,
                3e+0,
                1e+0,
                2e+0,
                3e+0
            ]
        }
    },
    "Task/parameters":{
        "iparam":{
            "ANA_SOL_BASIS":"ON",
            "ANA_SOL_PRINT_VIOLATED":"OFF",
            "AUTO_SORT_A_BEFORE_OPT":"OFF",
            "AUTO_UPDATE_SOL_INFO":"OFF",
            "BASIS_SOLVE_USE_PLUS_ONE":"OFF",
            "BI_CLEAN_OPTIMIZER":"OPTIMIZER_FREE",
            "BI_IGNORE_MAX_ITER":"OFF",
            "BI_IGNORE_NUM_ERROR":"OFF",
            "BI_MAX_ITERATIONS":1000000,
            "CACHE_LICENSE":"ON",
            "CHECK_CONVEXITY":"CHECK_CONVEXITY_FULL",
            "COMPRESS_STATFILE":"ON",
            "CONCURRENT_NUM_OPTIMIZERS":2,
            "CONCURRENT_PRIORITY_DUAL_SIMPLEX":2,
            "CONCURRENT_PRIORITY_FREE_SIMPLEX":3,
            "CONCURRENT_PRIORITY_INTPNT":4,
            "CONCURRENT_PRIORITY_PRIMAL_SIMPLEX":1,
            "FEASREPAIR_OPTIMIZE":"FEASREPAIR_OPTIMIZE_NONE",
            "INFEAS_GENERIC_NAMES":"OFF",
            "INFEAS_PREFER_PRIMAL":"ON",
            "INFEAS_REPORT_AUTO":"OFF",
            "INFEAS_REPORT_LEVEL":1,
            "INTPNT_BASIS":"BI_ALWAYS",
            "INTPNT_DIFF_STEP":"ON",
            "INTPNT_FACTOR_DEBUG_LVL":0,
```

```
            "INTPNT_FACTOR_METHOD":0,
            "INTPNT_HOTSTART":"INTPNT_HOTSTART_NONE",
            "INTPNT_MAX_ITERATIONS":400,
            "INTPNT_MAX_NUM_COR":-1,
            "INTPNT_MAX_NUM_REFINEMENT_STEPS":-1,
            "INTPNT_OFF_COL_TRH":40,
            "INTPNT_ORDER_METHOD":"ORDER_METHOD_FREE",
            "INTPNT_REGULARIZATION_USE":"ON",
            "INTPNT_SCALING":"SCALING_FREE",
            "INTPNT_SOLVE_FORM":"SOLVE_FREE",
            "INTPNT_STARTING_POINT":"STARTING_POINT_FREE",
            "LIC_TRH_EXPIRY_WRN":7,
            "LICENSE_DEBUG":"OFF",
            "LICENSE_PAUSE_TIME":0,
            "LICENSE_SUPPRESS_EXPIRE_WRNS":"OFF",
            "LICENSE_WAIT":"OFF",
            "LOG":10,
            "LOG_ANA_PRO":1,
            "LOG_BI":4,
            "LOG_BI_FREQ":2500,
            "LOG_CHECK_CONVEXITY":0,
            "LOG_CONCURRENT":1,
            "LOG_CUT_SECOND_OPT":1,
            "LOG_EXPAND":0,
            "LOG_FACTOR":1,
            "LOG_FEAS_REPAIR":1,
            "LOG_FILE":1,
            "LOG_HEAD":1,
            "LOG_INFEAS_ANA":1,
            "LOG_INTPNT":4,
            "LOG_MIO":4,
            "LOG_MIO_FREQ":1000,
            "LOG_OPTIMIZER":1,
            "LOG_ORDER":1,
            "LOG_PRESOLVE":1,
            "LOG_RESPONSE":0,
            "LOG_SENSITIVITY":1,
            "LOG_SENSITIVITY_OPT":0,
            "LOG_SIM":4,
            "LOG_SIM_FREQ":1000,
            "LOG_SIM_MINOR":1,
            "LOG_STORAGE":1,
            "MAX_NUM_WARNINGS":10,
            "MIO_BRANCH_DIR":"BRANCH_DIR_FREE",
            "MIO_CONSTRUCT_SOL":"OFF",
            "MIO_CUT_CLIQUE":"ON",
            "MIO_CUT_CMIR":"ON",
            "MIO_CUT_GMI":"ON",
            "MIO_CUT_KNAPSACK_COVER":"OFF",
            "MIO_HEURISTIC_LEVEL":-1,
            "MIO_MAX_NUM_BRANCHES":-1,
            "MIO_MAX_NUM_RELAXS":-1,
            "MIO_MAX_NUM_SOLUTIONS":-1,
            "MIO_MODE":"MIO_MODE_SATISFIED",
            "MIO_MT_USER_CB":"ON",
            "MIO_NODE_OPTIMIZER":"OPTIMIZER_FREE",
            "MIO_NODE_SELECTION":"MIO_NODE_SELECTION_FREE",
            "MIO_PERSPECTIVE_REFORMULATE":"ON",
            "MIO_PROBING_LEVEL":-1,
            "MIO_RINS_MAX_NODES":-1,
            "MIO_ROOT_OPTIMIZER":"OPTIMIZER_FREE",
            "MIO_ROOT_REPEAT_PRESOLVE_LEVEL":-1,
            "MT_SPINCOUNT":0,
```

```
"NUM_THREADS":0,
"OPF_MAX_TERMS_PER_LINE":5,
"OPF_WRITE_HEADER":"ON",
"OPF_WRITE_HINTS":"ON",
"OPF_WRITE_PARAMETERS":"OFF",
"OPF_WRITE_PROBLEM":"ON",
"OPF_WRITE_SOL_BAS":"ON",
"OPF_WRITE_SOL_ITG":"ON",
"OPF_WRITE_SOL_ITR":"ON",
"OPF_WRITE_SOLUTIONS":"OFF",
"OPTIMIZER":"OPTIMIZER_FREE",
"PARAM_READ_CASE_NAME":"ON",
"PARAM_READ_IGN_ERROR":"OFF",
"PRESOLVE_ELIMINATOR_MAX_FILL":-1,
"PRESOLVE_ELIMINATOR_MAX_NUM_TRIES":-1,
"PRESOLVE_LEVEL":-1,
"PRESOLVE_LINDEP_ABS_WORK_TRH":100,
"PRESOLVE_LINDEP_REL_WORK_TRH":100,
"PRESOLVE_LINDEP_USE":"ON",
"PRESOLVE_MAX_NUM_REDUCTIONS":-1,
"PRESOLVE_USE":"PRESOLVE_MODE_FREE",
"PRIMAL_REPAIR_OPTIMIZER":"OPTIMIZER_FREE",
"QO_SEPARABLE_REFORMULATION":"OFF",
"READ_DATA_COMPRESSED":"COMPRESS_FREE",
"READ_DATA_FORMAT":"DATA_FORMAT_EXTENSION",
"READ_DEBUG":"OFF",
"READ_KEEP_FREE_CON":"OFF",
"READ_LP_DROP_NEW_VARS_IN_BOU":"OFF",
"READ_LP_QUOTED_NAMES":"ON",
"READ_MPS_FORMAT":"MPS_FORMAT_FREE",
"READ_MPS_WIDTH":1024,
"READ_TASK_IGNORE_PARAM":"OFF",
"SENSITIVITY_ALL":"OFF",
"SENSITIVITY_OPTIMIZER":"OPTIMIZER_FREE_SIMPLEX",
"SENSITIVITY_TYPE":"SENSITIVITY_TYPE_BASIS",
"SIM_BASIS_FACTOR_USE":"ON",
"SIM_DEGEN":"SIM_DEGEN_FREE",
"SIM_DUAL_CRASH":90,
"SIM_DUAL_PHASEONE_METHOD":0,
"SIM_DUAL_RESTRICT_SELECTION":50,
"SIM_DUAL_SELECTION":"SIM_SELECTION_FREE",
"SIM_EXPLOIT_DUPVEC":"SIM_EXPLOIT_DUPVEC_OFF",
"SIM_HOTSTART":"SIM_HOTSTART_FREE",
"SIM_HOTSTART_LU":"ON",
"SIM_INTEGER":0,
"SIM_MAX_ITERATIONS":10000000,
"SIM_MAX_NUM_SETBACKS":250,
"SIM_NON_SINGULAR":"ON",
"SIM_PRIMAL_CRASH":90,
"SIM_PRIMAL_PHASEONE_METHOD":0,
"SIM_PRIMAL_RESTRICT_SELECTION":50,
"SIM_PRIMAL_SELECTION":"SIM_SELECTION_FREE",
"SIM_REFACTOR_FREQ":0,
"SIM_REFORMULATION":"SIM_REFORMULATION_OFF",
"SIM_SAVE_LU":"OFF",
"SIM_SCALING":"SCALING_FREE",
"SIM_SCALING_METHOD":"SCALING_METHOD_POW2",
"SIM_SOLVE_FORM":"SOLVE_FREE",
"SIM_STABILITY_PRIORITY":50,
"SIM_SWITCH_OPTIMIZER":"OFF",
"SOL_FILTER_KEEP_BASIC":"OFF",
"SOL_FILTER_KEEP_RANGED":"OFF",
"SOL_READ_NAME_WIDTH":-1,
```

```
        "SOL_READ_WIDTH":1024,
        "SOLUTION_CALLBACK":"OFF",
        "TIMING_LEVEL":1,
        "WRITE_BAS_CONSTRAINTS":"ON",
        "WRITE_BAS_HEAD":"ON",
        "WRITE_BAS_VARIABLES":"ON",
        "WRITE_DATA_COMPRESSED":0,
        "WRITE_DATA_FORMAT":"DATA_FORMAT_EXTENSION",
        "WRITE_DATA_PARAM":"OFF",
        "WRITE_FREE_CON":"OFF",
        "WRITE_GENERIC_NAMES":"OFF",
        "WRITE_GENERIC_NAMES_IO":1,
        "WRITE_IGNORE_INCOMPATIBLE_CONIC_ITEMS":"OFF",
        "WRITE_IGNORE_INCOMPATIBLE_ITEMS":"OFF",
        "WRITE_IGNORE_INCOMPATIBLE_NL_ITEMS":"OFF",
        "WRITE_IGNORE_INCOMPATIBLE_PSD_ITEMS":"OFF",
        "WRITE_INT_CONSTRAINTS":"ON",
        "WRITE_INT_HEAD":"ON",
        "WRITE_INT_VARIABLES":"ON",
        "WRITE_LP_FULL_OBJ":"ON",
        "WRITE_LP_LINE_WIDTH":80,
        "WRITE_LP_QUOTED_NAMES":"ON",
        "WRITE_LP_STRICT_FORMAT":"OFF",
        "WRITE_LP_TERMS_PER_LINE":10,
        "WRITE_MPS_FORMAT":"MPS_FORMAT_FREE",
        "WRITE_MPS_INT":"ON",
        "WRITE_PRECISION":15,
        "WRITE_SOL_BARVARIABLES":"ON",
        "WRITE_SOL_CONSTRAINTS":"ON",
        "WRITE_SOL_HEAD":"ON",
        "WRITE_SOL_IGNORE_INVALID_NAMES":"OFF",
        "WRITE_SOL_VARIABLES":"ON",
        "WRITE_TASK_INC_SOL":"ON",
        "WRITE_XML_MODE":"WRITE_XML_MODE_ROW"
    },
    "dparam":{
        "ANA_SOL_INFEAS_TOL":1e-6,
        "BASIS_REL_TOL_S":1e-12,
        "BASIS_TOL_S":1e-6,
        "BASIS_TOL_X":1e-6,
        "CHECK_CONVEXITY_REL_TOL":1e-10,
        "DATA_TOL_AIJ":1e-12,
        "DATA_TOL_AIJ_HUGE":1e+20,
        "DATA_TOL_AIJ_LARGE":1e+10,
        "DATA_TOL_BOUND_INF":1e+16,
        "DATA_TOL_BOUND_WRN":1e+8,
        "DATA_TOL_C_HUGE":1e+16,
        "DATA_TOL_CJ_LARGE":1e+8,
        "DATA_TOL_QIJ":1e-16,
        "DATA_TOL_X":1e-8,
        "FEASREPAIR_TOL":1e-10,
        "INTPNT_CO_TOL_DFEAS":1e-8,
        "INTPNT_CO_TOL_INFEAS":1e-10,
        "INTPNT_CO_TOL_MU_RED":1e-8,
        "INTPNT_CO_TOL_NEAR_REL":1e+3,
        "INTPNT_CO_TOL_PFEAS":1e-8,
        "INTPNT_CO_TOL_REL_GAP":1e-7,
        "INTPNT_NL_MERIT_BAL":1e-4,
        "INTPNT_NL_TOL_DFEAS":1e-8,
        "INTPNT_NL_TOL_MU_RED":1e-12,
        "INTPNT_NL_TOL_NEAR_REL":1e+3,
        "INTPNT_NL_TOL_PFEAS":1e-8,
        "INTPNT_NL_TOL_REL_GAP":1e-6,
```

```
        "INTPNT_NL_TOL_REL_STEP":9.95e-1,
        "INTPNT_QO_TOL_DFEAS":1e-8,
        "INTPNT_QO_TOL_INFEAS":1e-10,
        "INTPNT_QO_TOL_MU_RED":1e-8,
        "INTPNT_QO_TOL_NEAR_REL":1e+3,
        "INTPNT_QO_TOL_PFEAS":1e-8,
        "INTPNT_QO_TOL_REL_GAP":1e-8,
        "INTPNT_TOL_DFEAS":1e-8,
        "INTPNT_TOL_DSAFE":1e+0,
        "INTPNT_TOL_INFEAS":1e-10,
        "INTPNT_TOL_MU_RED":1e-16,
        "INTPNT_TOL_PATH":1e-8,
        "INTPNT_TOL_PFEAS":1e-8,
        "INTPNT_TOL_PSAFE":1e+0,
        "INTPNT_TOL_REL_GAP":1e-8,
        "INTPNT_TOL_REL_STEP":9.999e-1,
        "INTPNT_TOL_STEP_SIZE":1e-6,
        "LOWER_OBJ_CUT":-1e+30,
        "LOWER_OBJ_CUT_FINITE_TRH":-5e+29,
        "MIO_DISABLE_TERM_TIME":-1e+0,
        "MIO_MAX_TIME":-1e+0,
        "MIO_MAX_TIME_APRX_OPT":6e+1,
        "MIO_NEAR_TOL_ABS_GAP":0.0,
        "MIO_NEAR_TOL_REL_GAP":1e-3,
        "MIO_REL_GAP_CONST":1e-10,
        "MIO_TOL_ABS_GAP":0.0,
        "MIO_TOL_ABS_RELAX_INT":1e-5,
        "MIO_TOL_FEAS":1e-6,
        "MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT":0.0,
        "MIO_TOL_REL_GAP":1e-4,
        "MIO_TOL_X":1e-6,
        "OPTIMIZER_MAX_TIME":-1e+0,
        "PRESOLVE_TOL_ABS_LINDEP":1e-6,
        "PRESOLVE_TOL_AIJ":1e-12,
        "PRESOLVE_TOL_REL_LINDEP":1e-10,
        "PRESOLVE_TOL_S":1e-8,
        "PRESOLVE_TOL_X":1e-8,
        "QCQO_REFORMULATE_REL_DROP_TOL":1e-15,
        "SEMIDEFINITE_TOL_APPROX":1e-10,
        "SIM_LU_TOL_REL_PIV":1e-2,
        "SIMPLEX_ABS_TOL_PIV":1e-7,
        "UPPER_OBJ_CUT":1e+30,
        "UPPER_OBJ_CUT_FINITE_TRH":5e+29
    },
    "sparam":{
        "BAS_SOL_FILE_NAME":"",
        "DATA_FILE_NAME":"examples/tools/data/lo1.mps",
        "DEBUG_FILE_NAME":"",
        "INT_SOL_FILE_NAME":"",
        "ITR_SOL_FILE_NAME":"",
        "MIO_DEBUG_STRING":"",
        "PARAM_COMMENT_SIGN":"%%",
        "PARAM_READ_FILE_NAME":"",
        "PARAM_WRITE_FILE_NAME":"",
        "READ_MPS_BOU_NAME":"",
        "READ_MPS_OBJ_NAME":"",
        "READ_MPS_RAN_NAME":"",
        "READ_MPS_RHS_NAME":"",
        "SENSITIVITY_FILE_NAME":"",
        "SENSITIVITY_RES_FILE_NAME":"",
        "SOL_FILTER_XC_LOW":"",
        "SOL_FILTER_XC_UPR":"",
        "SOL_FILTER_XX_LOW":"",
```

```
            "SOL_FILTER_XX_UPR":"",
            "STAT_FILE_NAME":"",
            "STAT_KEY":"",
            "STAT_NAME":"",
            "WRITE_LP_GEN_VAR_NAME":"XMSKGEN"
        }
    }
}
```

14.8 The Solution File Format

MOSEK provides several solution files depending on the problem type and the optimizer used:

- *basis solution file* (extension `.bas`) if the problem is optimized using the simplex optimizer or basis identification is performed,
- *interior solution file* (extension `.sol`) if a problem is optimized using the interior-point optimizer and no basis identification is required,
- *integer solution file* (extension `.int`) if the problem contains integer constrained variables.

All solution files have the format:

```
NAME                : <problem name>
PROBLEM STATUS      : <status of the problem>
SOLUTION STATUS     : <status of the solution>
OBJECTIVE NAME      : <name of the objective function>
PRIMAL OBJECTIVE    : <primal objective value corresponding to the solution>
DUAL OBJECTIVE      : <dual objective value corresponding to the solution>
CONSTRAINTS
INDEX  NAME     AT ACTIVITY    LOWER LIMIT   UPPER LIMIT   DUAL LOWER   DUAL UPPER
?      <name>   ?? <a value>   <a value>     <a value>     <a value>    <a value>
VARIABLES
INDEX  NAME     AT ACTIVITY    LOWER LIMIT   UPPER LIMIT   DUAL LOWER   DUAL UPPER    CONIC␣
↪DUAL
?      <name>   ?? <a value>   <a value>     <a value>     <a value>    <a value>     <a value>
```

In the example the fields `?` and `<>` will be filled with problem and solution specific information. As can be observed a solution report consists of three sections, i.e.

- `HEADER` In this section, first the name of the problem is listed and afterwards the problem and solution status are shown. Next the primal and dual objective values are displayed.
- `CONSTRAINTS` For each constraint i of the form

$$l_i^c \le \sum_{j=1}^{n} a_{ij} x_j \le u_i^c, \tag{14.10}$$

  the following information is listed:

  - `INDEX`: A sequential index assigned to the constraint by **MOSEK**
  - `NAME`: The name of the constraint assigned by the user.
  - `AT`: The status of the constraint. In Table 14.4 the possible values of the status keys and their interpretation are shown.

Table 14.4: Status keys.

Status key	Interpretation
UN	Unknown status
BS	Is basic
SB	Is superbasic
LL	Is at the lower limit (bound)
UL	Is at the upper limit (bound)
EQ	Lower limit is identical to upper limit
**	Is infeasible i.e. the lower limit is greater than the upper limit.

- ACTIVITY: the quantity $\sum_{j=1}^{n} a_{ij}x_j^*$, where x^* is the value of the primal solution.
- LOWER LIMIT: the quantity l_i^c (see (14.10).)
- UPPER LIMIT: the quantity u_i^c (see (14.10).)
- DUAL LOWER: the dual multiplier corresponding to the lower limit on the constraint.
- DUAL UPPER: the dual multiplier corresponding to the upper limit on the constraint.

- VARIABLES The last section of the solution report lists information about the variables. This information has a similar interpretation as for the constraints. However, the column with the header CONIC DUAL is included for problems having one or more conic constraints. This column shows the dual variables corresponding to the conic constraints.

Example: lo1.sol

In Listing 14.7 we show the solution file for the lo1.opf problem.

Listing 14.7: An example of .sol file.

```
NAME                : 
PROBLEM STATUS      : PRIMAL_AND_DUAL_FEASIBLE
SOLUTION STATUS     : OPTIMAL
OBJECTIVE NAME      : obj
PRIMAL OBJECTIVE    : 8.33333333e+01
DUAL OBJECTIVE      : 8.33333332e+01

CONSTRAINTS
INDEX      NAME          AT ACTIVITY                 LOWER LIMIT        UPPER LIMIT        ␣
↪DUAL LOWER               DUAL UPPER
0          c1            EQ 3.00000000000000e+01     3.00000000e+01     3.00000000e+01     -0.
↪00000000000000e+00     -2.49999999741654e+00
1          c2            SB 5.33333333049188e+01     1.50000000e+01     NONE               2.
↪09157603759397e-10      -0.00000000000000e+00
2          c3            UL 2.49999999842049e+01     NONE               2.50000000e+01     -0.
↪00000000000000e+00     -3.33333332895110e-01

VARIABLES
INDEX      NAME          AT ACTIVITY                 LOWER LIMIT        UPPER LIMIT        ␣
↪DUAL LOWER               DUAL UPPER
0          x1            LL 1.67020427073508e-09     0.00000000e+00     NONE               -4.
↪49999999528055e+00     -0.00000000000000e+00
1          x2            LL 2.93510446280504e-09     0.00000000e+00     1.00000000e+01     -2.
↪16666666494916e+00     6.20863861687316e-10
2          x3            SB 1.49999999899425e+01     0.00000000e+00     NONE               -8.
↪79123177454657e-10     -0.00000000000000e+00
3          x4            SB 8.33333332273116e+00     0.00000000e+00     NONE               -1.
↪69795978899185e-09     -0.00000000000000e+00
```

CHAPTER

FIFTEEN

INTERFACE CHANGES

The section show interface-specific changes to the **MOSEK** Fusion API for Python in version 8. See the release notes for general changes and new features of the **MOSEK** Optimization Suite.

15.1 Compatibility

Fusion API has undergo a deep refactorization that will most likely make old code fail to compile. On a general level:

- more linear operators are avaible,
- pretty printing is implemented for most classes,
- variable operators (such slicingm stacking,...) are now moved to a specific class *Var*, pretty much like expressions have their own *Expr*.
- dimensions can now be expressed directly with arrays instead of the *Set* class
- reduced need for explicit conversion from variable to expression, i.e. the *Variable.asExpr*,
- new syntax to specify integer variables, as well as a short-hand for binary ones.

Compatibility guarantees for this interface has been updated. See the new *state of compatibility*.

15.2 Parameters

Added

- *intpntQoTolDfeas*
- *intpntQoTolInfeas*
- *intpntQoTolMuRed*
- *intpntQoTolNearRel*
- *intpntQoTolPfeas*
- *intpntQoTolRelGap*
- *semidefiniteTolApprox*
- *intpntMultiThread*
- *licenseTrhExpiryWrn*
- *logAnaPro*
- *mioCutClique*
- *mioCutGmi*

- *mioCutImpliedBound*
- *mioCutKnapsackCover*
- *mioCutSelectionLevel*
- *mioPerspectiveReformulate*
- *mioRootRepeatPresolveLevel*
- *mioVbDetectionLevel*
- *presolveEliminatorMaxFill*

Removed

- feasrepairTol
- mioHeuristicTime
- mioMaxTimeAprxOpt
- mioRelAddCutLimited
- mioTolMaxCutFracRhs
- mioTolMinCutFracRhs
- mioTolRelRelaxInt
- mioTolX
- nonconvexTolFeas
- nonconvexTolOpt
- allocAddQnz
- concurrentNumOptimizers
- concurrentPriorityDualSimplex
- concurrentPriorityFreeSimplex
- concurrentPriorityIntpnt
- concurrentPriorityPrimalSimplex
- feasrepairOptimize
- intpntFactorDebugLvl
- intpntFactorMethod
- licTrhExpiryWrn
- logConcurrent
- logNonconvex
- logParam
- logSimNetworkFreq
- mioBranchPrioritiesUse
- mioContSol
- mioCutCg
- mioCutLevelRoot
- mioCutLevelTree
- mioFeaspumpLevel

- mioHotstart
- mioKeepBasis
- mioLocalBranchNumber
- mioOptimizerMode
- mioPresolveAggregate
- mioPresolveProbing
- mioPresolveUse
- mioStrongBranch
- mioUseMultithreadedOptimizer
- nonconvexMaxIterations
- presolveElimFill
- presolveEliminatorUse
- qoSeparableReformulation
- readAnz
- readCon
- readCone
- readMpsKeepInt
- readMpsObjSense
- readMpsRelax
- readQnz
- readVar
- warningLevel
- writeIgnoreIncompatibleConicItems
- writeIgnoreIncompatibleNlItems
- writeIgnoreIncompatiblePsdItems
- feasrepairNamePrefix
- feasrepairNameSeparator
- feasrepairNameWsumviol

15.3 Constants

Added

Changed

Removed

- beginConcurrent
- beginNetworkDualSimplex
- beginNetworkPrimalSimplex

- beginNetworkSimplex
- beginNonconvex
- beginSimplexNetworkDetect
- endConcurrent
- endNetworkDualSimplex
- endNetworkPrimalSimplex
- endNetworkSimplex
- endNonconvex
- endSimplexNetworkDetect
- imMioPresolve
- imNetworkDualSimplex
- imNetworkPrimalSimplex
- imNonconvex
- noncovex
- updateNetworkDualSimplex
- updateNetworkPrimalSimplex
- updateNonconvex
- concurrentTime
- mioCgSeperationTime
- mioCmirSeperationTime
- simNetworkDualTime
- simNetworkPrimalTime
- simNetworkTime
- ptom
- ptox
- concurrentFastestOptimizer
- mioNumBasisCuts
- mioNumCardgubCuts
- mioNumCoefRedcCuts
- mioNumContraCuts
- mioNumDisaggCuts
- mioNumFlowCoverCuts
- mioNumGcdCuts
- mioNumGubCoverCuts
- mioNumKnapsurCoverCuts
- mioNumLatticeCuts
- mioNumLiftCuts
- mioNumObjCuts
- mioNumPlanLocCuts

- simNetworkDualDegIter
- simNetworkDualHotstart
- simNetworkDualHotstartLu
- simNetworkDualInfIter
- simNetworkDualIter
- simNetworkPrimalDegIter
- simNetworkPrimalHotstart
- simNetworkPrimalHotstartLu
- simNetworkPrimalInfIter
- simNetworkPrimalIter
- solIntProsta
- solIntSolsta
- stoNumACacheFlushes
- stoNumATransposes
- lazy
- concurrent
- mixedIntConic
- networkPrimalSimplex
- nonconvex
- primalDualSimplex

BIBLIOGRAPHY

[AA95] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Math. Programming*, 71(2):221–245, 1995.

[AGMX96] E. D. Andersen, J. Gondzio, Cs. Mészáros, and X. Xu. Implementation of interior point methods for large scale linear programming. In T. Terlaky, editor, *Interior-point methods of mathematical programming*, pages 189–252. Kluwer Academic Publishers, 1996.

[ART03] E. D. Andersen, C. Roos, and T. Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Math. Programming*, February 2003.

[AY96] E. D. Andersen and Y. Ye. Combining interior-point and pivoting algorithms. *Management Sci.*, 42(12):1719–1731, December 1996.

[And09] Erling D. Andersen. The homogeneous and self-dual model and algorithm for linear optimization. Technical Report TR-1-2009, MOSEK ApS, 2009. URL: http://docs.mosek.com/whitepapers/homolo.pdf.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[GJ79] Michael R Gary and David S Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979.

[GY05] Donald Goldfarb and Wotao Yin. Second-order cone programming methods for total variation-based image restoration. *SIAM Journal on Scientific Computing*, 27(2):622–645, 2005.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[GK00] Richard C. Grinold and Ronald N. Kahn. *Active portfolio management*. McGraw-Hill, New York, 2 edition, 2000.

[Naz87] J. L. Nazareth. *Computer Solution of Linear Programs*. Oxford University Press, New York, 1987.

[PB15] Jaehyun Park and Stephen Boyd. A semidefinite programming method for integer convex quadratic minimization. *arXiv preprint arXiv:1504.07672*, 2015.

[Pat03] Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM review*, 45(1):116–123, 2003.

[Wol98] L. A. Wolsey. *Integer programming*. John Wiley and Sons, 1998.

[BenTalN01] A. Ben-Tal and A. Nemirovski. *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. MPS/SIAM Series on Optimization. SIAM, 2001.

[MOSEKApS12] MOSEK ApS. *The MOSEK Modeling Cookbook*. MOSEK ApS, Fruebjergvej 3, Boks 16, 2100 Copenhagen O, 2012. Last revised September 2015. URL: http://docs.mosek.com/generic/modeling-a4.pdf.

Classes

Enumerations

Exceptions

Parameters

Response codes