MOSEK Fusion API for Python

Release 9.0.98

MOSEK ApS

15 July 2019

Contents

Chapter 1

Introduction

The **MOSEK** Optimization Suite 9.0.98 is a powerful software package capable of solving large-scale optimization problems of the following kind:

- linear,
- conic:
  - conic quadratic (also known as second-order cone),
  - involving the exponential cone,
  - involving the power cone,
  - semidefinite,
- convex quadratic and quadratically constrained,
- integer.

In order to obtain an overview of features in the **MOSEK** Optimization Suite consult the product introduction guide.

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 advantages of linear optimization may outweigh 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

$$Ax - b \geq 0.$$

In conic optimization this is replaced with a wider class of constraints

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

where $\mathcal{K}$ is a *convex cone*. For example in 3 dimensions $\mathcal{K}$ may correspond to an ice cream cone. The conic optimizer in **MOSEK** supports a number of 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.

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 modeling instead of coding.

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. *Fusion* always yields readable and easily portable code.

The Fusion API for Python provides access to Conic Optimization, including:

- Linear Optimization (LO)
- Conic Quadratic (Second-Order Cone) Optimization (CQO, SOCO)
- Power Cone Optimization
- Conic Exponential Optimization (CEO)
- Semidefinite Optimization (SDO)
- Mixed-Integer Optimization (MIO)

as well as to an auxiliary linear algebra library.

Convex Quadratic and Quadratically Constrained (QCQO) problems can be reformulated as Conic Quadratic problems and subsequently solved using *Fusion*. This is the recommended approach, as described in the **MOSEK** Modeling Cookbook and this whitepaper.

Chapter 2

Contact Information

Phone	+45 7174 9373	
Website	mosek.com	
Email		
	sales@mosek.com	Sales, pricing, and licensing
	support@mosek.com	Technical support, questions and bug reports
	info@mosek.com	Everything else.
Mailing Address		
	MOSEK ApS	
	Fruebjergvej 3	
	Symbion Science Park, Box 16	
	2100 Copenhagen O	
	Denmark	

You can get in touch with **MOSEK** using popular social media as well:

Blogger	https://blog.mosek.com/
Google Group	https://groups.google.com/forum/#!forum/mosek
Twitter	https://twitter.com/mosektw
Google+	https://plus.google.com/+Mosek/posts
Linkedin	https://www.linkedin.com/company/mosek-aps

In particular **Twitter** is used for news, updates and release announcements.

Chapter 3

License Agreement

Before using the **MOSEK** software, please read the license agreement available in the distribution at `<MSKHOME>/mosek/9.0/mosek-eula.pdf` or on the **MOSEK** website https://mosek.com/products/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 3.1.

Listing 3.1: *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:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

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 3.2.

Listing 3.2: *fplib* license.

```
/****************************************************************
 *
```

(continues on next page)

(continued from previous page)

```
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

Zstandard

MOSEK includes the *Zstandard* library developed by Facebook obtained from github/zstd. The license agreement for *Zstandard* is shown in Listing 3.3.

Listing 3.3: *Zstandard* license.

```
BSD License

For Zstandard software

Copyright (c) 2016-present, Facebook, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name Facebook nor the names of its contributors may be used to
  endorse or promote products derived from this software without specific
  prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Chapter 4

Installation

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

Important: Before running this **MOSEK** interface please make sure that you:

- Installed **MOSEK** correctly. Some operating systems require extra steps. See the Installation guide for instructions and common troubleshooting tips.
- Set up a license. See the Licensing guide for instructions.

Compatibility

The Fusion API for Python requires Python with numpy. The supported versions of Python are shown below:

Platform	Python	PyPy
Linux 64 bit	2.7, 3.6+	2.7
Mac OS 64 bit	2.7, 3.6+	2.7
Windows 32 and 64 bit	2.7, 3.6+	2.7

4.1 Anaconda

The **MOSEK** Optimization Suite can be installed as an Anaconda package, see https://anaconda.org/MOSEK/mosek, for example by running

```
conda install -c mosek mosek
```

If you installed the **MOSEK** package as part of Anaconda, no additional setup is required.

4.2 PIP and Wheels

The **MOSEK** Optimization Suite can be installed as a Wheels package with PIP, using

```
pip install -f https://download.mosek.com/stable/wheel/index.html Mosek --user
```

(skip `--user` for a system-wide installation).

If you installed the **MOSEK** package with PIP, no additional setup is required.

4.3 PyPy

To use **MOSEK** in PyPy install the **MOSEK** Python module from the directory `<PLATFORM>/purepython` instead of `<PLATFORM>/python` as described below.

4.4 Manual installation

Locating files in the MOSEK Optimization Suite

The relevant files of the Fusion API for Python are organized as reported in Table 4.1.

Table 4.1: Relevant files for the Fusion API for Python.

Relative Path	Description	Label
`<MSKHOME>/mosek/9.0/tools/platform/<PLATFORM>/python/2`	Python 2 install	`<PYTHON2DIR>`
`<MSKHOME>/mosek/9.0/tools/platform/<PLATFORM>/python/3`	Python 3 install	`<PYTHON3DIR>`
`<MSKHOME>/mosek/9.0/tools/examples/fusion/python`	Examples	`<EXDIR>`
`<MSKHOME>/mosek/9.0/tools/examples/fusion/data`	Additional data	`<MISCDIR>`

where

- `<MSKHOME>` is the folder in which the **MOSEK** Optimization Suite has been installed,
- `<PLATFORM>` is the actual platform among those supported by **MOSEK**, i.e. `win32x86`, `win64x86`, `linux64x86` or `osx64x86`.

Manual install and setting up paths

To install **MOSEK** for Python run the `<PYTHON2DIR>/setup.py` or `<PYTHON3DIR>/setup.py` script depending on the Python version you want to use. This will add the **MOSEK** module to your Python distribution's library of modules. The script accepts the standard options typical for Python setup scripts. For instance, to install **MOSEK** for Python 3 in the user's local library run:

```
$ python3 <PYTHON3DIR>/setup.py install --user
```

on Linux and Mac OS or

```
C:\> python3 <PYTHON3DIR>\setup.py install --user
```

on Windows.

For a system-wide installation drop the `--user` flag.

4.5 Testing the Installation

First of all, to check that the Fusion API for Python was properly installed, start Python and try

```
import mosek
```

The installation can further be tested by running some of the enclosed examples. Open a terminal, change folder to `<EXDIR>` and use Python to run a selected example, for instance:

```
python lo1.py
```

Chapter 5

Design Overview

Fusion is a result of many years of experience in conic optimization. It is a dedicated API for users who want to enjoy a simpler experience interfacing with the solver. This applies to users who regularly solve conic problems, and to new users who do not want to be too bothered with the technicalities of a low-level optimizer. *Fusion* is designed for fast and clean prototyping of conic problems without suffering excessive performance degradation.

Note that *Fusion* **is** an object-oriented framework for conic-optimization but it **is not** a general purpose modeling language. The main design principles of *Fusion* are:

- **Expressiveness**: we try to make it nice! Despite not being a modeling language, *Fusion* yields readable, easy to maintain code that closely resembles the mathematical formulation of the problem.
- **Seamlessly multi-language** : *Fusion* code can be ported across C++, Python, Java, .NET and with only minimal adaptations to the syntax of each language.
- **What you write is what MOSEK gets**: A *Fusion* model is fed into the solver with (almost) no additional transformations.

Expressiveness

Suppose you have a conic quadratic optimization problem like the efficient frontier in portfolio optimization:

$$\begin{array}{ll}\text{maximize} & \mu^T x - \alpha\gamma \\ \text{subject to} & e^T x = w, \\ & \gamma \geq \|G^T x\|, \\ & x \geq 0.\end{array}$$

Its representation in *Fusion* is a direct translation of the mathematical model:

```
M.objective(ObjectiveSense.Maximize, Expr.sub(Expr.dot(mu, x), Expr.mul(alpha, gamma)))

M.constraint(Expr.sub(Expr.sum(x), w), Domain.equalsTo(0.0))
M.constraint(Expr.vstack(gamma, Expr.mul(G.transpose(), x)), Domain.inQCone())
M.constraint(x, Domain.greaterThan(0.0))
```

Seamless multi-language API

Fusion can easily be ported across the five supported languages. All functionalities and naming conventions remain the same in all of them. This has some advantages:

- Simplifies code sharing between developers working in different languages.
- Improves code reusability.
- Simplifies the transition from R&D to production (for instance from fast-prototyping languages used in R&D to more efficient ones used for high performance).

Here is the same code snippet (creation of a variable in the model) in all languages supported by *Fusion*. Careful code design can generate models with only the necessary syntactic differences between implementations.

```
auto x= M->variable("x", 3, Domain::greaterThan(0.0));        // C++
```

```
x = M.variable('x', 3, Domain.greaterThan(0.0))               # Python
```

```
Variable x = M.variable("x", 3, Domain.greaterThan(0.0))      // Java
```

```
Variable x = M.Variable("x", 3, Domain.GreaterThan(0.0))      // C#
```

What You Write is What MOSEK Gets

Fusion is not a modeling language. Instead it clearly defines the formulation the user must adhere to and only provides functionalities required for that formulation. An important upshot is that *Fusion* will not modify the problem provided by the user, except for introducing auxiliary variables required to fit the problem into the format of the low-level optimizer API. In other words, the problem that is actually solved is as close as possible to what the user writes.

For example, suppose the user defined a conic constraint

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

Now the low-level API requires that all variables appearing in all conic constraints are different, and so *Fusion* will have to replace the conic constraint with

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = y,$$
$$y_1 \geq \sqrt{y_2^2 + y_3^2}.$$

Note, however, that to use the optimizer API directly the user would have to apply the same transformation! A similar situation happens when the user defines a number of linear constraints, which have to be arranged into a large linear constraint matrix A, and so on. So, in effect, the *Fusion* mechanism only automates operations that the user would have to carry out anyway (using pencil and paper, presumably). Otherwise the optimizer model is a direct copy of the *Fusion* model.

The main benefits of this approach are:

- The user knows what problem is actually being solved.
- Dual information is readily available for all variables and constraints.
- Only the necessary overhead.
- Better control over numerical stability.

Chapter 6

Conic Modeling

6.1 The model

A model built using *Fusion* is **always** a conic optimization problem and it is convex by definition. These problems can be succinctly characterized as

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + b \in \mathcal{K} \end{array} \tag{6.1}$$

where $\mathcal{K}$ is a product of the following basic types of cones:

- *linear:* $\mathbb{R}$, $\mathbb{R}_+$, $\{0\}$,
- *quadratic:* $\mathcal{Q}^n = \{x \in \mathbb{R}^n \ : \ x_1 \geq \sqrt{x_2^2 + \cdots + x_n^2}\}$,
- *rotated quadratic:* $\mathcal{Q}_r^n = \{x \in \mathbb{R}^n \ : \ 2x_1x_2 \geq x_3^2 + \cdots + x_n^2, \ x_1, x_2 \geq 0\}$,
- *primal power cone:* $\mathcal{P}_n^{\alpha,1-\alpha} = \{x \in \mathbb{R}^n \ : \ x_1^\alpha x_2^{1-\alpha} \geq \sqrt{x_3^2 + \cdots + x_n^2}, \ x_1, x_2 \geq 0\}$, or its dual,
- *primal exponential:* $K_{\exp} = \{x \in \mathbb{R}^3 \ : \ x_1 \geq x_2 \exp(x_3/x_2), \ x_1, x_2 \geq 0\}$, or its dual,
- *semidefinite::* $\mathcal{S}_+^n = \{X \in \mathbb{R}^{n\times n} \ : \ X \text{ is symmmetric positive semidefinite}\}$.

The main thing about a *Fusion* model is that it can be specified in a convenient way without explicitly constructing the representation (6.1). Instead the user has access to *variables* which are used to construct *linear operators* that appear in *constraints*. The cone types described above are the domains of those constraints. A *Fusion* model can potentially contain many different building blocks of that kind. To facilitate manipulations with a large number of variables *Fusion* defines various *logical views* of parts of the model.

This section briefly summarizes the constructions and techniques available in *Fusion*. See Sec. 7 for a basic tutorial and Sec. 11 for more advanced case studies. This section is only an introduction: detailed specification of the methods and classes mentioned here can be found in the *API reference*.

A *Fusion* model is represented by the class `Model` and created by a simple construction

```
M = Model()
M = Model('modelName')

with Model() as M:
```

The model object is the user's interface to the optimization problem, used in particular for

- formulating the problem by defining variables, constraints and objective,
- solving the problem and retrieving the solution status and solutions,
- interacting with the solver: setting up parameters, registering for callbacks, performing I/O, obtaining detailed information from the optimizer etc.
- memory management.

Almost all elements of the model: variables, constraints and the model itself can be constructed with or without names. If used, the names for each type of object must be unique. Choosing a good naming convention can make the problem more readable when dumped to a file.

6.2 Variables

Continuous variables can be scalars, vectors or higher-dimensional arrays. The are added to the model with the method *Model.variable* which returns a representing object of type *Variable*. The shape of a variable (number of dimensions and length in each dimension) has to be specified at creation. Optionally a variable may be created in a restricted domain (by default variables are unbounded, that is in $\mathbb{R}$). For instance, to declare a variable $x \in \mathbb{R}^n_+$ we could write

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

A multi-dimensional variable is declared by specifying an array with all dimension sizes. Here is an $n \times n$ variable:

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

The specification of dimensions can also be part of the domain, as in this declaration of a symmetric positive semidefinite variable of dimension n:

```
v = M.variable(Domain.inPSDCone(n));
```

Integer variables are specified with an additional domain modifier. To add an integer variable $z \in [1, 10]$ we write

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

The function *Domain.binary* is a shorthand for binary variables often appearing in combinatorial problems:

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

Integrality requirement can be switched on and off using the methods *Variable.makeInteger* and *Variable.makeContinuous*.

A domain usually allows to specify the number of objects to be created. For example here is a definition of m symmetric positive semidefinite variables of dimension n each. The actual variable `x` will be of shape $m \times n \times n$ where each slice with fixed first coordinate is an $n \times n$ PSD:

```
x = M.variable(Domain.inPSDCone(n, m))
```

The *Variable* object provides the primal (*Variable.level*) and dual (*Variable.dual*) solution values of the variable after optimization, and it enters in the construction of linear expressions involving the variable.

6.3 Linear algebra

Linear expressions are constructed combining *variables* and *matrices* by linear operators. The result is an object that represents the linear expression itself. *Fusion* only allows for those combinations of operators and arguments that yield linear functions of the variables. Expressions have shapes and dimensions in the same fashion as variables. For instance, if $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$, then Ax is a vector expression of length m. Note, however, that the internal size of Ax is mn, because each entry is a linear combination for which m coefficients have to be stored.

Expressions are concrete implementations of the virtual interface *Expression*. In typical situations, however, all operations on expressions can be performed using the static methods and factory methods of the class *Expr*.

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 or matrix-scalar multiplication
Expr.neg	Sign inversion
Expr.outer	Vector outer-product
Expr.dot	Dot product
Expr.sum	Sum over a given dimension
Expr.mulElm	Element-wise multiplication
Expr.mulDiag	Sum over the diagonal of a matrix which is the result of a matrix multiplication
Expr.constTerm	Return a *constant term*

Operations on expressions must adhere to the rules of matrix algebra regarding dimensions; otherwise a *DimensionError* exception will be thrown.

Expression can be composed, nested and used as building blocks in new expressions. For instance $Ax + By$ can be implemented as:

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

For operations involving multiple variables and expressions the users should consider list-based methods. For instance, a clean way to write $x + y + z + w$ would be:

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

Note that a single variable (object of class *Variable*) can also be used as an expression. Once constructed, expressions are immutable.

6.4 Constraints and objective

Constraints are declared within an optimization model using the method *Model.constraint*. Every constraint in *Fusion* has the form

Expression belongs to a *Domain*.

Objects of type *Domain* correspond roughly to the types of convex cones $\mathcal{K}$ mentioned at the beginning of this section. For instance, the following set of linear constraints

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

could 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))
```

Note that the scalar domain *Domain.equalsTo* consisting of a single point 0 scales up to the dimension of the expression and applies to all its elements. This allows many constraints to be comfortably expressed in a vectorized form. See also Sec. 6.7.

The *Constraint* object provides the dual (*Constraint.dual*) value of the constraint after optimization and the primal value of the constraint expression (*Constraint.level*).

The typical domains used to specify constraints are listed below. Note that they can also be used directly at variable creation, whenever that makes sense.

	Type	Domain
Linear	equality	*Domain.equalsTo*
	inequality $\leq$	*Domain.lessThan*
	inequality $\geq$	*Domain.greaterThan*
	two-sided bound	*Domain.inRange*
Conic Quadratic	quadratic cone	*Domain.inQCone*
	rotated quadratic cone	*Domain.inRotatedQCone*
Other Conic	exponential cone	*Domain.inPExpCone*
	power cone	*Domain.inPPowerCone* (α)
Semidefinite	PSD matrix	*Domain.inPSDCone*
Integral	Integers in domain D	*Domain.integral* (D)
	$\{0,1\}$	*Domain.binary*

Having discussed variables and constraints we can finish by defining the optimization objective with *Model.objective*. The objective function is a scalar expression and the objective sense is specified by the enumeration *ObjectiveSense* as either *minimize* or *maximize*. The typical linear objective function $c^T x$ can be declared as

```
M.objective( ObjectiveSense.Minimize,  Expr.dot(c,x) )
```

6.5 Matrices

At some point it becomes necessary to specify linear expressions such as Ax where A is a (large) constant data matrix. Such coefficient matrices can be represented in dense or sparse format. Dense matrices can always be represented using the standard data structures for arrays and two-dimensional arrays built into the language. Alternatively, or when sparsity can be exploited, matrices can be constructed as objects of the class *Matrix*. This can have some advantages: a more generic code that can be ported across platforms and can be used with *both* dense and sparse matrices without modifications.

Dense matrices are constructed with a variant of the static factory method *Matrix.dense*. The values of all entries must be specified all at once and the resulting matrix is immutable. For example the matrix

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

can be defined with:

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

or from a flattened representation:

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

Sparse matrices are constructed with a variant of the static factory method *Matrix.sparse*. This is both speed- and memory-efficient when the matrix has few nonzero entries. A matrix A in sparse format is given by a list of triples (i, j, v), each defining one entry: $A_{i,j} = v$. The order does not matter. The entries not in the list are assumed to be 0. For example, take the matrix

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.0 \\ 0.0 & 3.0 & 0.0 & 4.0 \end{bmatrix}.$$

Assuming we number rows and columns from 0, the corresponding list of triplets is:

$$A = \{(0, 0, 1.0), (0, 3, 2.0), (1, 1, 3.0), (1, 3, 4.0)\}$$

The *Fusion* definition would be:

```
rows  = [ 0, 0, 1, 1 ]
cols  = [ 0, 3, 1, 3 ]
values= [ 1.0, 2.0, 3.0, 4.0 ]

m = Matrix.sparse(len(rows), len(cols), rows, cols, values)
```

The *Matrix* class provides more standard constructions such as the identity matrix, a constant value matrix, block diagonal matrices etc.

6.6 Stacking and views

Fusion provides a way to construct logical views of parts of existing expressions or combinations of existing expressions. They are still represented by objects of type *Variable* or *Expression* that refer to the original ones. This can be useful in some scenarios:

- retrieving only the values of a few variables, and ignoring the remaining auxiliary ones,
- stacking vectors or matrices to perform various matrix operations,
- bundling a number of similar constraints into one; see Sec. 6.7,
- adding constraints between parts of the same variable, etc.

All these operations do not require *new* variables or expressions, but just lightweight *logical views*. In what follows we will concentrate on expressions; the same techniques are available for variables. These techniques will be familiar to the users of numerical tools such as Matlab or NumPy.

Picking and slicing

Expression.pick *picks* a subset of entries from a variable or expression. Special cases of picking are *Expression.index*, which picks just one scalar entry and *Expression.slice* which picks a *slice*, that is restricts each dimension to a subinterval. Slicing is a frequently used operation.

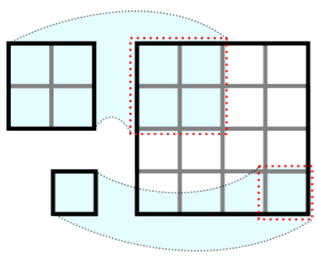


Fig. 6.1: Two dimensional slicing.

Both displayed regions are slices of the two-dimensional 4×4 expression, which can be selected as follows:

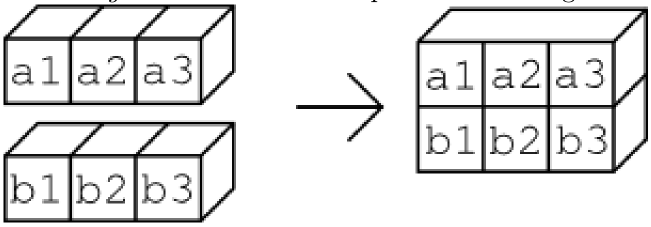
```
A1 = Ax.slice([0,0],[2,2])
A2 = Ax.index([3,3])
```

Reshaping

Expressions can be *reshaped* creating a view with the same number of coordinates arranged in a different way. A particular example of this operation if *flattening*, which converts any multi-dimensional expression into a one-dimensional vector.

Stacking

Stacking refers to the concatenation of expressions to form a new larger one. For example, the next figure depicts the *vertical stacking* of two vectors of shape 1×3 resulting in a matrix of shape 2×3.




```
c = Expr.vstack([a, b]);
```

Vertical stacking (*Expr.vstack*) of expressions of shapes $d_1 \times d_2$ and $d_1' \times d_2$ has shape $(d_1 + d_1') \times d_2$. Similarly, *horizontal stacking* (*Expr.hstack*) of expressions of shapes $d_1 \times d_2$ and $d_1 \times d_2'$ has shape $d_1 \times (d_2 + d_2')$. *Fusion* supports also more general versions of stacking for multi-dimensional variables, as described in *Expr.stack*. A special case of stacking is *repetition* (*Expr.repeat*), equivalent to stacking copies of the same expression.

6.7 Vectorization

Using *Fusion* one can compactly express sequences of similar constraints. For example, if we want to express

$$Ax_i = b_i, \ i = 1, \dots, n$$

we can think of $x_i \in \mathbb{R}^m, b_i \in \mathbb{R}^k$ as the columns of two matrices $X = [x_1, \dots, x_n] \in \mathbb{R}^{m \times n}$, $B = [b_1, \dots, b_n] \in \mathbb{R}^{k \times n}$, and write simply

$$AX - B = 0.$$

```
X = Var.hstack( [ xi[i] for i in range(n) ] )
B = Expr.hstack( [ bi[i] for i in range(n) ] )

M.constraint(Expr.sub(Expr.mul(A, X), B), Domain.equalsTo(0.0))
```

In this example the domain *Domain.equalsTo* scales to apply to all the entries of the expression.

Another powerful case of vectorization and scaling domains is the ability to define a sequence of conic constraints in one go. Suppose we want to find an upper bound on the 2-norm of a sequence of vectors, that is we want to express

$$t \geq \|y_i\|, \ i = 1, \dots, n$$

Suppose that the vectors y_i are arranged in the rows of a matrix Y. Then we can simply write:

```
t = M.variable();

M.constraint(Expr.hstack(Var.vrepeat(t, n), Y), Domain.inQCone())
```

Here, again, the conic domain *Domain.inQCone* is by default applied to each row of the matrix separately, yielding the desired constraints in a loop-free way (the i-th row is (t, y_i)). The direction along which conic constraints are created within multi-dimensional expressions can be changed with *Domain.axis*.

We recommend vectorizing the code whenever possible. It is not only more elegant and portable but also more efficient — loops are eliminated and the number of *Fusion* API calls is reduced.

6.8 Reoptimization

Between optimizations the user can modify the model in a few ways:

- Add new constraints with *Model.constraint*. This is useful for solving a sequence of optimization problems with more and more restrictions on the feasible set. See for example Sec. 11.8.
- Add new variables with *Model.variable*.
- Replace the objective with a new one. This is particularly useful when solving a sequence of problems with the same data but different objectives, for instance in multi-objective optimization. For simplicity, suppose we want to minimize $f(x) = \gamma x + \beta y$, for varying choices of $\gamma > 0$. Then we could write:

```
gamma=[0., 0.5, 1.0]                    # Choices for gamma
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()
```

- Update part of the objective (*Model.updateObjective*).
- Update an existing constraint or replace the constraint expression with a new one (*Constraint.update*).

Otherwise all *Fusion* objects are immutable. See also Sec. 7.9 for an example.

Chapter 7

Optimization Tutorials

In this section we demonstrate how to set up basic types of optimization problems. Each short tutorial contains a working example of formulating problems, defining variables and constraints and retrieving solutions.

7.1 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

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

The problem description consists of the following elements:

- m and n — the number of constraints and variables, respectively,
- x — the variable vector of length n,
- c — the coefficient vector of length n

$$c = \left[\begin{array}{c} c_0 \\ \vdots \\ c_{n-1} \end{array} \right],$$

- c^f — fixed term in the objective,
- A — an $m \times n$ matrix of coefficients

$$A = \left[\begin{array}{ccc} a_{0,0} & \cdots & a_{0,(n-1)} \\ \vdots & \cdots & \vdots \\ a_{(m-1),0} & \cdots & a_{(m-1),(n-1)} \end{array} \right],$$

- l^c and u^c — the lower and upper bounds on constraints,
- l^x and u^x — the lower and upper bounds on variables.

Please note that we are using 0 as the first index: x_0 is the first element in variable vector x.

The *Fusion* user does not need to specify all of the above elements explicitly — they will be assembled from the *Fusion* model.

7.1.1 Example LO1

The following is an example of a small linear optimization problem:

$$
\begin{array}{lccccccccl}
\text{maximize} & 3x_0 & + & 1x_1 & + & 5x_2 & + & 1x_3 & & \\
\text{subject to} & 3x_0 & + & 1x_1 & + & 2x_2 & & & = & 30, \\
 & 2x_0 & + & 1x_1 & + & 3x_2 & + & 1x_3 & \geq & 15, \\
 & & & 2x_1 & & & + & 3x_3 & \leq & 25,
\end{array}
\tag{7.1}
$$

under 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:
```

For this simple problem we are going to enter all the linear coefficients directly:

```
    A = [[3.0, 1.0, 2.0, 0.0],
         [2.0, 1.0, 3.0, 1.0],
         [0.0, 2.0, 0.0, 3.0]]
    c = [3.0, 1.0, 5.0, 1.0]
```

The variables appearing in problem (7.1) can be declared as one 4-dimensional variable:

```
        x = M.variable("x", 4, Domain.greaterThan(0.0))
```

At this point we already have variables with bounds $0 \leq x_i \leq \infty$, because the domain is applied element-wise to the entries of the variable vector. Next, we impose the upper bound on x_1:

```
        M.constraint(x.index(1), Domain.lessThan(10.0))
```

The linear constraints can now be entered one by one using the dot product of our variable with a coefficient vector:

```
        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 in the same way:

```
        M.objective("obj", ObjectiveSense.Maximize, Expr.dot(c, x))
```

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

```
        M.solve()
```

The solution values can be attained with the method *Variable.level*.

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

Listing 7.1: *Fusion* implementation of model (7.1).

```
from mosek.fusion import *

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

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

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

        # Create constraints
        M.constraint(x.index(1), Domain.lessThan(10.0))
        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)]))
```

7.2 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. Since the set $\mathbb{R}^n$ of real numbers is also a convex cone, we can simply write a compound conic constraint $x \in \mathcal{K}$ where $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_l$ is a product of smaller cones and x is the full problem variable.

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

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

In this tutorial we describe how to use the two types of quadratic cones defined as:

- 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:

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n : 2x_0x_1 \geq \sum_{j=2}^{n-1} x_j^2, \quad x_0 \geq 0, \quad x_1 \geq 0 \right\}.$$

For other types of cones supported by **MOSEK** see Sec. 7.3, Sec. 7.4, Sec. 7.5. See *Domain* for a list and definitions of available cone types. Different cone types can appear together in one optimization problem.

For example, the following constraint:

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

describes a convex cone in $\mathbb{R}^3$ given by the inequality:

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

In *Fusion* the coordinates of a cone are not restricted to single variables. They can be arbitrary linear expressions, and an auxiliary variable will be substituted by *Fusion* in a way transparent to the user.

7.2.1 Example CQO1

Consider the following conic quadratic problem which involves some linear constraints, a quadratic cone and a rotated quadratic cone.

$$\begin{array}{ll}
\text{minimize} & y_1 + y_2 + y_3 \\
\text{subject to} & x_1 + x_2 + 2.0x_3 = 1.0, \\
 & x_1, x_2, x_3 \geq 0.0, \\
 & (y_1, x_1, x_2) \in \mathcal{Q}^3, \\
 & (y_2, y_3, x_3) \in \mathcal{Q}_r^3.
\end{array} \tag{7.2}$$

We start by creating the optimization model:

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

We then define variables x and y. Two logical variables (aliases) z1 and z2 are introduced to model the quadratic cones. These are not new variables, but map onto parts of x and y for the sake of convenience.

```
    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))
```

The linear constraint is defined using the dot product:

```
    # 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* created previously. Note that this is a basic way of defining conic constraints, and that in practice they would have more complicated structure.

```
    # 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.
```

(continues on next page)

(continued from previous page)

```
    #                 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())
```

We only need the objective function:

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

Calling the *Model.solve* method invokes the solver:

```
    M.solve()
    M.writeTask('cqo1.opf')
```

The primal and dual solution values can be retrieved using *Variable.level*, *Constraint.level* and *Variable.dual*, *Constraint.dual*, respectively:

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

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

Listing 7.2: *Fusion* implementation of model (7.2).

```
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()
    M.writeTask('cqo1.opf')
```

(continues on next page)

(continued from previous page)

```
# Get the linear solution 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))
```

7.3 Power Cone 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. Since the set $\mathbb{R}^n$ of real numbers is also a convex cone, we can simply write a compound conic constraint $x \in \mathcal{K}$ where $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_l$ is a product of smaller cones and x is the full problem variable.

MOSEK can solve conic 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

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

In this tutorial we describe how to use the power cone. The primal power cone of dimension n with parameter $0 < \alpha < 1$ is defined as:

$$\mathcal{P}_n^{\alpha,1-\alpha} = \left\{ x \in \mathbb{R}^n : x_0^\alpha x_1^{1-\alpha} \geq \sqrt{\sum_{i=2}^{n-1} x_i^2},\ x_0, x_1 \geq 0 \right\}.$$

In particular, the most important special case is the three-dimensional power cone family:

$$\mathcal{P}_3^{\alpha,1-\alpha} = \left\{ x \in \mathbb{R}^3 : x_0^\alpha x_1^{1-\alpha} \geq |x_2|,\ x_0, x_1 \geq 0 \right\}.$$

For example, the conic constraint $(x, y, z) \in \mathcal{P}_3^{0.25,0.75}$ is equivalent to $x^{0.25} y^{0.75} \geq |z|$, or simply $xy^3 \geq z^4$ with $x, y \geq 0$.

MOSEK also supports the dual power cone:

$$\left(\mathcal{P}_n^{\alpha,1-\alpha}\right)^* = \left\{ x \in \mathbb{R}^n : \left(\frac{x_0}{\alpha}\right)^\alpha \left(\frac{x_1}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{i=2}^{n-1} x_i^2},\ x_0, x_1 \geq 0 \right\}.$$

For other types of cones supported by **MOSEK** see Sec. 7.2, Sec. 7.4, Sec. 7.5. See `Domain` for a list and definitions of available cone types. Different cone types can appear together in one optimization problem.

In *Fusion* the coordinates of a cone are not restricted to single variables. They can be arbitrary linear expressions, and an auxiliary variable will be substituted by *Fusion* in a way transparent to the user.

7.3.1 Example POW1

Consider the following optimization problem which involves powers of variables:

$$\begin{array}{ll} \text{maximize} & x^{0.2}y^{0.8} + z^{0.4} - x \\ \text{subject to} & x + y + \frac{1}{2}z = 2, \\ & x, y, z \geq 0. \end{array} \tag{7.3}$$

With $(x, y, z) = (x_0, x_1, x_2)$ we convert it into conic form using auxiliary variables as bounds for the power expressions:

$$\begin{array}{lrcl} \text{maximize} & x_3 + x_4 - x_0 & & \\ \text{subject to} & x_0 + x_1 + \frac{1}{2}x_2 & = & 2, \\ & (x_0, x_1, x_3) & \in & \mathcal{P}_3^{0.2,0.8}, \\ & (x_2, x_5, x_4) & \in & \mathcal{P}_3^{0.4,0.6}, \\ & x_5 & = & 1. \end{array} \tag{7.4}$$

We start by creating the optimization model:

```
with Model('pow1') as M:
```

We then define the variable `x` corresponding to the original problem (7.3), and auxiliary variables appearing in the conic reformulation (7.4).

```
    x  = M.variable('x', 3, Domain.unbounded())
    x3 = M.variable()
    x4 = M.variable()
```

The linear constraint is defined using the dot product operator *Expr.dot*:

```
    # Create the linear constraint
    M.constraint(Expr.dot(x, [1.0, 1.0, 0.5]), Domain.equalsTo(2.0))
```

The primal power cone is referred to via *Domain.inPPowerCone* with an appropriate list of variables or expressions in each case.

```
    # Create the power cone constraints
    M.constraint(Var.vstack(x.slice(0,2), x3), Domain.inPPowerCone(0.2))
    M.constraint(Expr.vstack(x.index(2), 1.0, x4), Domain.inPPowerCone(0.4))
```

We only need the objective function:

```
    # Set the objective function
    M.objective(ObjectiveSense.Maximize, Expr.dot([1.0,1.0,-1.0], Var.vstack(x3, x4, x.
↪index(0))))
```

Calling the *Model.solve* method invokes the solver:

```
    M.solve()
```

The primal and dual solution values can be retrieved using *Variable.level*, *Constraint.level* and *Variable.dual*, *Constraint.dual*. Here we just display the primal solution

```
    # Get the linear solution values
    solx = x.level()
    print('x,y,z = %s' % str(solx))
```

which is

```
[ 0.06389298  0.78308564  2.30604283 ]
```

Listing 7.3: *Fusion* implementation of model (7.3).

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

(continues on next page)

(continued from previous page)

```
with Model('pow1') as M:

    x  = M.variable('x', 3, Domain.unbounded())
    x3 = M.variable()
    x4 = M.variable()

    # Create the linear constraint
    M.constraint(Expr.dot(x, [1.0, 1.0, 0.5]), Domain.equalsTo(2.0))

    # Create the power cone constraints
    M.constraint(Var.vstack(x.slice(0,2), x3), Domain.inPPowerCone(0.2))
    M.constraint(Expr.vstack(x.index(2), 1.0, x4), Domain.inPPowerCone(0.4))

    # Set the objective function
    M.objective(ObjectiveSense.Maximize, Expr.dot([1.0,1.0,-1.0], Var.vstack(x3, x4, x.
↪index(0))))

    # Solve the problem
    M.solve()

    # Get the linear solution values
    solx = x.level()
    print('x,y,z = %s' % str(solx))
```

7.4 Conic Exponential 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. Since the set $\mathbb{R}^n$ of real numbers is also a convex cone, we can simply write a compound conic constraint $x \in \mathcal{K}$ where $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_l$ is a product of smaller cones and x is the full problem variable.

MOSEK can solve conic optimization problems of the form

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

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

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

In this tutorial we describe how to use the primal exponential cone defined as:

$$K_{\exp} = \left\{ x \in \mathbb{R}^3 : x_0 \geq x_1 \exp(x_2/x_1),\ x_0, x_1 \geq 0 \right\}.$$

MOSEK also supports the dual exponential cone:

$$K_{\exp}^* = \left\{ s \in \mathbb{R}^3 : s_0 \geq -s_2 e^{-1} \exp(s_1/s_2),\ s_2 \leq 0, s_0 \geq 0 \right\}.$$

For other types of cones supported by **MOSEK** see Sec. 7.2, Sec. 7.3, Sec. 7.5. See *Domain* for a list and definitions of available cone types. Different cone types can appear together in one optimization problem.

For example, the following constraint:

$$(x_4, x_0, x_2) \in K_{\exp}$$

describes a convex cone in $\mathbb{R}^3$ given by the inequalities:

$$x_4 \geq x_0 \exp(x_2/x_0),\ \ x_0, x_4 \geq 0.$$

In *Fusion* the coordinates of a cone are not restricted to single variables. They can be arbitrary linear expressions, and an auxiliary variable will be substituted by *Fusion* in a way transparent to the user.

7.4.1 Example CEO1

Consider the following basic conic exponential problem which involves some linear constraints and an exponential inequality:

$$\begin{array}{lrcl} \mbox{minimize} & x_0 + x_1 & & \\ \mbox{subject to} & x_0 + x_1 + x_2 & = & 1, \\ & x_0 & \geq & x_1 \exp(x_2/x_1), \\ & x_0, x_1 & \geq & 0. \end{array} \tag{7.5}$$

The conic form of (7.5) is:

$$\begin{array}{lrcl} \mbox{minimize} & x_0 + x_1 & & \\ \mbox{subject to} & x_0 + x_1 + x_2 & = & 1, \\ & (x_0, x_1, x_2) & \in & K_{\exp}, \\ & x & \in & \mathbb{R}^3. \end{array} \tag{7.6}$$

We start by creating the optimization model:

```
with Model('ceo1') as M:
```

We then define the variable x.

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

The linear constraint is defined using the sum operator *Expr.sum*:

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

The conic exponential constraint in this case is very simple as it involves just the variable x. The primal exponential cone is referred to via *Domain.inPExpCone*, and it must be applied to a variable of length 3 or an array of such variables. Note that this is a basic way of defining conic constraints, and that in practice they would have more complicated structure.

```
    # Create the conic exponential constraint
    expc = M.constraint("expc", x, Domain.inPExpCone())
```

We only need the objective function:

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

Calling the *Model.solve* method invokes the solver:

```
    M.solve()
```

The primal and dual solution values can be retrieved using *Variable.level*, *Constraint.level* and *Variable.dual*, *Constraint.dual*, respectively:

```
    # Get the linear solution values
    solx = x.level()
```

```
    # Get conic solution of expc
    expcval  = expc.level()
    expcdual = expc.dual()
```

Listing 7.4: *Fusion* implementation of model (7.5).

```
from mosek.fusion import *

with Model('ceo1') as M:

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

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

    # Create the conic exponential constraint
    expc = M.constraint("expc", x, Domain.inPExpCone())

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

    # Solve the problem
    M.solve()

    M.writeTask('ceo1.ptf')
    # Get the linear solution values
    solx = x.level()
    print('x1,x2,x3 = %s' % str(solx))

    # Get conic solution of expc
    expcval  = expc.level()
    expcdual = expc.dual()
    print('expc levels                = %s' % str(expcval))
    print('expc dual conic var levels = %s' % str(expcdual))
```

7.5 Semidefinite Optimization

Semidefinite optimization is a generalization of conic 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

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

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

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

In *Fusion* the user can enter the linear expressions in a more convenient way, without having to cast the problem exactly in the above form.

7.5.1 Example SDO1

We consider the simple optimization problem with semidefinite and conic quadratic constraints:

$$
\begin{array}{llll}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \overline{X} \right\rangle + x_0 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_0 & = & 1, \\
 & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_1 + x_2 & = & 1/2, \\
 & x_0 \geq \sqrt{{x_1}^2 + {x_2}^2}, & \overline{X} \succeq 0, &
\end{array}
\tag{7.7}
$$

The problem description contains a 3-dimensional symmetric semidefinite variable which can be written explicitly as:

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

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

$$
\begin{array}{rcl}
\overline{X}_{00} + \overline{X}_{11} + \overline{X}_{22} + x_0 & = & 1, \\
\overline{X}_{00} + \overline{X}_{11} + \overline{X}_{22} + 2(\overline{X}_{10} + \overline{X}_{20} + \overline{X}_{21}) + x_1 + x_2 & = & 1/2.
\end{array}
$$

Our implementation in *Fusion* begins with creating a new model:

```
    with Model("sdo1") as M:
```

We create a symmetric semidefinite variable $\overline{X}$ and another variable representing x. For simplicity we immediately declare that x belongs to a quadratic cone

```
        X = M.variable("X", Domain.inPSDCone(3))
        x = M.variable("x", Domain.inQCone(3))
```

In this elementary example we are going to create an explicit matrix representation of the problem

$$
\overline{C} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \ \overline{A}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \ \overline{A}_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.
$$

and use it in the model via the dot product operation $\langle \cdot, \cdot \rangle$ which applies to matrices as well as to vectors. This way we create each of the linear constraints and the objective as one expression.

```
        # 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))
```

Now it remains to solve the problem with *Model.solve*.

Listing 7.5: *Fusion* implementation of problem (7.7).

```
import mosek
from   mosek.fusion import *

def main(args):
    with Model("sdo1") as M:

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

        # Setting up constant coefficient matrices
        C  = Matrix.dense ( [[2.,1.,0.],[1.,2.,1.],[0.,1.,2.]] )
        A1 = Matrix.eye(3)
        A2 = Matrix.ones(3,3)

        # 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.solve()

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

7.6 Integer Optimization

An optimization problem where one or more of the variables are constrained to integer values is called a (mixed) integer optimization problem. **MOSEK** supports integer variables in combination with linear, quadratic and quadratically constrtained and conic problems (except semidefinite). See the previous tutorials for an introduction to how to model these types of problems.

7.6.1 Example MILO1

We use the example

$$\begin{array}{lccl} \text{maximize} & x_0 + 0.64 x_1 & & \\ \text{subject to} & 50 x_0 + 31 x_1 & \leq & 250, \\ & 3 x_0 - 2 x_1 & \geq & -4, \\ & x_0, x_1 \geq 0 & & \text{and integer} \end{array} \tag{7.8}$$

to demonstrate how to set up and solve a problem with integer variables. It has the structure of a linear optimization problem (see Sec. 7.1) except for integrality constraints on the variables. Therefore, only the specification of the integer constraints requires something new compared to the linear optimization problem discussed previously.

First, the integrality constraints are imposed by modifying any existing domain with *Domain.integral*:

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

Another way to do this is to use the method *Variable.makeInteger* on a selected variable.

Next, the example demonstrates how to set various useful parameters of the mixed-integer optimizer. See Sec. 13.4 for details.

```
        # 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)
```

The complete source for the example is listed in Listing 7.6.

Listing 7.6: How to solve problem (7.8).

```
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
        print('[x0, x1] = ', x.level())
        print("MIP rel gap = %.2f (%f)" % (M.getSolverDoubleInfo(
            "mioObjRelGap"), M.getSolverDoubleInfo("mioObjAbsGap")))
```

7.6.2 Specifying an initial solution

It is a common strategy to provide a starting feasible point (if one is known in advance) to the mixed-integer solver. This can in many cases reduce solution time.

It is not necessary to specify the whole solution. **MOSEK** will attempt to use it to speed up the computation. **MOSEK** will first try to construct a feasible solution by fixing integer variables to the values provided by the user (rounding if necessary) and optimizing over the continuous variables. The outcome of this process can be inspected via information items *"mioConstructSolution"* and *"mioConstructSolutionObj"*, and via the `Construct solution objective` entry in the log. We concentrate on a simple example below.

$$\begin{array}{ll} \text{maximize} & 7x_0 + 10x_1 + x_2 + 5x_3 \\ \text{subject to} & x_0 + x_1 + x_2 + x_3 \leq 2.5 \\ & x_0, x_1, x_2 \in \mathbb{Z} \\ & x_0, x_1, x_2, x_3 \geq 0 \end{array} \tag{7.9}$$

Solution values can be set using *Variable.setLevel* .

Listing 7.7: Implementation of problem (7.9) specifying an initial solution.

```
        # Assign values to integer variables.
        # We only set a slice of xx
        init_sol = [1.0, 1.0, 0.0]
        x.slice(0,3).setLevel(init_sol)
```

A more advanced application of *Variable.setLevel* is presented in the case study on *Multiprocessor scheduling*.

The log output from the optimizer will in this case indicate that the inputted values were used to construct an initial feasible solution:

```
Construct solution objective       : 1.950000000000e+01
```

The same information can be obtained from the API:

Listing 7.8: Retrieving information about usage of initial solution

```
        constr = M.getSolverIntInfo("mioConstructSolution")
        constrVal = M.getSolverDoubleInfo("mioConstructSolutionObj")
        print("Initial solution utilization: {0}\nInitial solution objective: {1:.3f}\n".
↪format(constr, constrVal))
```

7.6.3 Example MICO1

Integer variables can also be used arbitrarily in conic problems (except semidefinite). We refer to the previous tutorials for how to set up a conic optimization problem. Here we present sample code that sets up a simple optimization problem:

$$\begin{array}{ll} \text{minimize} & x^2+y^2 \\ \text{subject to} & x \geq e^y + 3.8, \\ & x, y \text{ integer}. \end{array} \tag{7.10}$$

The canonical conic formulation of (7.10) suitable for Fusion API for Python is

$$\begin{array}{lll} \text{minimize} & t & \\ \text{subject to} & (t,x,y) \in \mathcal{Q}^3 & (t \geq \sqrt{x^2+y^2}) \\ & (x-3.8, 1, y) \in K_{\exp} & (x-3.8 \geq e^y) \\ & x, y \text{ integer}, & \\ & t \in \mathbb{R}. & \end{array} \tag{7.11}$$

Listing 7.9: Implementation of problem (7.11).

```
from mosek.fusion import *

with Model('mico1') as M:

    x = M.variable(Domain.integral(Domain.unbounded()))
    y = M.variable(Domain.integral(Domain.unbounded()))
    t = M.variable()

    M.constraint(Expr.vstack(t, x, y), Domain.inQCone())
    M.constraint(Expr.vstack(Expr.sub(x, 3.8), 1, y), Domain.inPExpCone())

    M.objective(ObjectiveSense.Minimize, t)

    M.setLogHandler(sys.stdout)
    M.solve()

    print('Solution: x = {0}, y = {1}'.format(x.level()[0], y.level()[0]))
```

Error and solution status handling were omitted for readability.

7.7 Geometric Programming

Geometric programs (GP) are a particular class of optimization problems which can be expressed in special polynomial form as positive sums of generalized monomials. More precisely, a geometric problem in canonical form is

$$\begin{array}{lll} \text{minimize} & f_0(x) & \\ \text{subject to} & f_i(x) \leq 1, & i = 1, \ldots, m, \\ & x_j > 0, & j = 1, \ldots, n, \end{array} \tag{7.12}$$

where each $f_0, \ldots, f_m$ is a *posynomial*, that is a function of the form

$$f(x) = \sum_k c_k x_1^{\alpha_{k1}} x_2^{\alpha_{k2}} \cdots x_n^{\alpha_{kn}}$$

with arbitrary real α_{ki} and $c_k > 0$. The standard way to formulate GPs in convex form is to introduce a variable substitution

$$x_i = \exp(y_i).$$

Under this substitution all constraints in a GP can be reduced to the form

$$\log(\sum_k \exp(a_k^T y + b_k)) \leq 0 \tag{7.13}$$

involving a *log-sum-exp* bound. Moreover, constraints involving only a single monomial in x can be even more simply written as a linear inequality:

$$a_k^T y + b_k \leq 0$$

We refer to the **MOSEK** Modeling Cookbook and to *[BKVH07]* for more details on this reformulation. A geometric problem formulated in convex form can be entered into **MOSEK** with the help of exponential cones.

7.7.1 Example GP1

The following problem comes from *[BKVH07]*. Consider maximizing the volume of a $h \times w \times d$ box subject to upper bounds on the area of the floor and of the walls and bounds on the ratios h/w and d/w:

$$\begin{array}{lrl} \text{maximize} & hwd & \\ \text{subject to} & 2(hw + hd) & \leq A_{\text{wall}}, \\ & wd & \leq A_{\text{floor}}, \\ & \alpha & \leq h/w \leq \beta, \\ & \gamma & \leq d/w \leq \delta. \end{array} \tag{7.14}$$

The decision variables in the problem are h, w, d. We make a substitution

$$h = \exp(x), w = \exp(y), d = \exp(z)$$

after which (7.14) becomes

$$\begin{array}{ll} \text{maximize} & x + y + z \\ \text{subject to} & \log(\exp(x + y + \log(2/A_{\text{wall}})) + \exp(x + z + \log(2/A_{\text{wall}}))) \leq 0, \\ & y + z \leq \log(A_{\text{floor}}), \\ & \log(\alpha) \leq x - y \leq \log(\beta), \\ & \log(\gamma) \leq z - y \leq \log(\delta). \end{array} \tag{7.15}$$

Next, we demonstrate how to implement a log-sum-exp constraint (7.13). It can be written as:

$$\begin{array}{l} u_k \geq \exp(a_k^T y + b_k), \quad (\text{equiv. } (u_k, 1, a_k^T y + b_k) \in K_{\text{exp}}), \\ \sum_k u_k = 1. \end{array} \tag{7.16}$$

This presentation requires one extra variable u_k for each monomial appearing in the original posynomial constraint.

Listing 7.10: Implementation of log-sum-exp as in (7.16).

```
# Models log(sum(exp(Ax+b))) <= 0.
# Each row of [A b] describes one of the exp-terms
def logsumexp(M, A, x, b):
    k = int(A.shape[0])
    u = M.variable(k)
    M.constraint(Expr.sum(u), Domain.equalsTo(1.0))
    M.constraint(Expr.hstack(u,
                             Expr.constTerm(k, 1.0),
                             Expr.add(Expr.mul(A, x), b)), Domain.inPExpCone())
```

We can now use this function to assemble all constraints in the model. The linear part of the problem is entered as in Sec. 7.1.

Listing 7.11: Source code solving problem (7.15).

```
def max_volume_box(Aw, Af, alpha, beta, gamma, delta):
    with Model('max_vol_box') as M:
        xyz = M.variable(3)
        M.objective('Objective', ObjectiveSense.Maximize, Expr.sum(xyz))

        logsumexp(M, array([[1,1,0],[1,0,1]]), xyz, array([log(2.0/Aw), log(2.0/Aw)]))

        M.constraint(Expr.dot([0, 1, 1], xyz), Domain.lessThan(log(Af)))
        M.constraint(Expr.dot([1,-1, 0], xyz), Domain.inRange(log(alpha),log(beta)))
        M.constraint(Expr.dot([0,-1, 1], xyz), Domain.inRange(log(gamma),log(delta)))

        M.setLogHandler(sys.stdout)
        M.solve()

        return exp(xyz.level())
```

Given sample data we obtain the solution h, w, d as follows:

Listing 7.12: Sample data for problem (7.14).

```
Aw, Af, alpha, beta, gamma, delta = 200.0, 50.0, 2.0, 10.0, 2.0, 10.0
h,w,d = max_volume_box(Aw, Af, alpha, beta, gamma, delta)
print("h={0:.3f}, w={1:.3f}, d={2:.3f}".format(h, w, d))
```

7.8 Library of basic functions

This section contains a library of small models of basic functions frequently appearing in optimization models. It is essentially an implementation of the mathematical models from the **MOSEK** Modeling Cookbook using Fusion API for Python. These short code snippets can be seen as illustrative examples, can be copy-pasted to other code, and can even be directly called when assembling optimization models as we show in Sec. 7.8.6 (although this may be more suitable for prototyping; also note that additional variables and constraints will be introduced and there is no error checking).

7.8.1 Variable and constraint management

Variable duplication

$x = y$

Listing 7.13: Duplicate variables.

```
# x = y
def dup(M, x, y):
    M.constraint(Expr.sub(x,y), Domain.equalsTo(0.0))
```

7.8.2 Linear operations

Absolute value

$t \geq |x|$

Listing 7.14: Absolute value.

```
# t >= |x|, where t, x have the same shape
def abs(M, t, x):
    M.constraint(Expr.add(t,x), Domain.greaterThan(0.0))
    M.constraint(Expr.sub(t,x), Domain.greaterThan(0.0))
```

1-norm

$t \geq \sum_i |x_i|$

Listing 7.15: 1-norm.

```
# t >= sum( |x_i| ), x is a vector expression
def norm1(M, t, x):
    u = M.variable(x.getShape(), Domain.unbounded())
    abs(M, u, x)
    M.constraint(Expr.sub(t, Expr.sum(u)), Domain.greaterThan(0.0))
```

7.8.3 Quadratic and power operations

Square

$t \geq x^2$

Listing 7.16: Square.

```
# t >= x^2
def sq(M, t, x):
    M.constraint(Expr.hstack(0.5, t, x), Domain.inRotatedQCone())
```

2-norm

$t \geq \sqrt{\sum_i x_i^2}$

Listing 7.17: 2-norm.

```
# t >= sqrt(x_1^2 + ... + x_n^2) where x is a vector
def norm2(M, t, x):
    M.constraint(Expr.vstack(t, x), Domain.inQCone())
```

Powers

$t \geq |x|^p$, $p > 1$

Listing 7.18: Power.

```
# t >= |x|^p (where p>1)
def pow(M, t, x, p):
    M.constraint(Expr.hstack(t, 1, x), Domain.inPPowerCone(1.0/p))
```

$t \geq 1/x^p, \ x > 0, p > 0$

Listing 7.19: Power reciprocal.

```
# t >= 1/|x|^p, x>0 (where p>0)
def pow_inv(M, t, x, p):
    M.constraint(Expr.hstack(t, x, 1), Domain.inPPowerCone(1.0/(1.0+p)))
```

p-norm

$t \geq (\sum_i |x_i|^p)^{1/p}$, $p > 1$

Listing 7.20: p-norm.

```
# t >= \|x\|_p (where p>1), x is a vector expression
def pnorm(M, t, x, p):
    n = int(x.getSize())
    r = M.variable(n)
    M.constraint(Expr.sub(t, Expr.sum(r)), Domain.equalsTo(0.0))
    M.constraint(Expr.hstack(Var.repeat(t,n), r, x), Domain.inPPowerCone(1.0-1.0/p))
```

Geometric mean

$t \leq (x_1 \cdot \dots \cdot x_n)^{1/n}$, $x_i > 0$

Listing 7.21: Geometric mean.

```
# |t| <= (x_1...x_n)^(1/n), x_i>=0, x is a vector expression of length >= 1
def geo_mean(M, t, x):
    n = int(x.getSize())
    if n==1:
        abs(M, x, t)
    else:
        t2 = M.variable()
        M.constraint(Expr.hstack(t2, x.index(n-1), t), Domain.inPPowerCone(1.0-1.0/n))
        geo_mean(M, t2, x.slice(0,n-1))
```

7.8.4 Exponentials and logarithms

log

$t \leq \log x$, $x > 0$

Listing 7.22: Logarithm.

```
# t <= log(x), x>=0
def log(M, t, x):
    M.constraint(Expr.hstack(x, 1, t), Domain.inPExpCone())
```

exp

$t \geq e^x$

Listing 7.23: Exponential.

```
# t >= exp(x)
def exp(M, t, x):
    M.constraint(Expr.hstack(t, 1, x), Domain.inPExpCone())
```

Entropy

$t \geq x \log x$, $x > 0$

Listing 7.24: Entropy.

```
# t >= x * log(x), x>=0
def ent(M, t, x):
    M.constraint(Expr.hstack(1, x, Expr.neg(t)), Domain.inPExpCone())
```

Relative entropy

$t \geq x \log x/y, \ x, y > 0$

Listing 7.25: Relative entropy.

```
# t >= x * log(x/y), x,y>=0
def relent(M, t, x, y):
    M.constraint(Expr.hstack(y, x, Expr.neg(t)), Domain.inPExpCone())
```

Log-sum-exp

$\log \sum_i e^{x_i} \leq t$

Listing 7.26: Log-sum-exp.

```
# log( sum_i(exp(x_i)) ) <= t, where x is a vector
def logsumexp(M, t, x):
    n = int(x.getSize())
    u = M.variable(n)
    M.constraint(Expr.hstack(u, Expr.constTerm(n, 1.0), Expr.sub(x, Var.repeat(t, n))), Domain.
↪inPExpCone())
    M.constraint(Expr.sum(u), Domain.lessThan(1.0))
```

7.8.5 Integer Modeling

Semicontinuous variable

$x \in \{0\} \cup [a, b], \, b > a > 0$

Listing 7.27: Semicontinuous variable.

```
# x = 0 or a <= x <= b
def semicontinuous(M, x, a, b):
    u = M.variable(x.getShape(), Domain.binary())
    M.constraint(Expr.sub(x, Expr.mul(a, u)), Domain.greaterThan(0.0))
    M.constraint(Expr.sub(x, Expr.mul(b, u)), Domain.lessThan(0.0))
```

Indicator variable

$x \neq 0 \implies t = 1$. We assume x is a priori normalized so $|x_i| \leq 1$.

Listing 7.28: Indicator variable.

```
# x!=0 implies t=1. Assumes that |x|<=1 in advance.
def indicator(M, t, x):
    M.constraint(t, Domain.inRange(0,1))
    t.makeInteger()
    abs(M, t, x)
```

Logical OR

At least one of the conditions is true.

Listing 7.29: Logical OR.

```
# x OR y, where x, y are binary
def logic_or(M, x, y):
    M.constraint(Expr.add(x, y), Domain.greaterThan(1.0))
# x_1 OR ... OR x_n, where x is a binary vector
def logic_or_vect(M, x):
    M.constraint(Expr.sum(x), Domain.greaterThan(1.0))
```

Logical NAND

At most one of the conditions is true (also known as SOS1).

Listing 7.30: Logical NAND.

```
# at most one of x_1,...,x_n, where x is a binary vector (SOS1 constraint)
def logic_sos1(M, x):
    M.constraint(Expr.sum(x), Domain.lessThan(1.0))
# NOT(x AND y), where x, y are binary
def logic_nand(M, x, y):
    M.constraint(Expr.add(x, y), Domain.lessThan(1.0))
```

Cardinality bound

At most k of the continuous variables are nonzero. We assume x is a priori normalized so $|x_i| \leq 1$.

Listing 7.31: Cardinality bound.

```
# At most k of entries in x are nonzero, assuming in advance |x_i|<=1.
def card(M, x, k):
    t = M.variable(x.getShape(), Domain.binary())
    abs(M, t, x)
    M.constraint(Expr.sum(t), Domain.lessThan(k))
```

7.8.6 Model assembly example

We now demonstrate how to quickly build a simple optimization model for the problem

$$\begin{array}{ll} \text{maximize} & -\sqrt{x^2+y^2}+\log y - x^{1.5}, \\ \text{subject to} & x \geq y+3, \end{array} \tag{7.17}$$

or equivalently

$$\begin{array}{ll} \text{maximize} & -t_0+t_1-t_2, \\ \text{subject to} & x \geq y+3, \\ & t_0 \geq \sqrt{x^2+y^2}, \\ & t_1 \leq \log y, \\ & t_2 \geq x^{1.5}. \end{array}$$

Listing 7.32: Modeling (7.17).

```
def testExample():
    M = Model()
    x = M.variable()
    y = M.variable()
    t = M.variable(3)

    M.constraint(Expr.sub(x, y), Domain.greaterThan(3.0))
    norm2(M, t.index(0), Var.vstack(x,y))
```

(continues on next page)

(continued from previous page)

```
    log  (M, t.index(1), y)
    pow  (M, t.index(2), x, 1.5)

    M.objective(ObjectiveSense.Maximize, Expr.dot(t, [-1,1,-1]))
```

7.9 Problem Modification and Reoptimization

Often one might want to solve not just a single optimization problem, but a sequence of problems, each differing only slightly from the previous one. This section demonstrates how to modify and re-optimize an existing problem. The example we study is a simple production planning model.

Problem modifications regarding variables, cones, objective function and constraints can be grouped in categories:

- adding constraints and variables,
- modifying existing constraints.

Adding new variables and constraints is very easy. Modifications to existing constraints are more cumbersome, and the user should consider whether it is not worth rebuilding the model from scratch in such case. The amount of work required by *Fusion* to update the optimizer task may outweigh the potential gains.

Depending on the type of modification, **MOSEK** may be able to optimize the modified problem more efficiently exploiting the information and internal state from the previous execution. After optimization, the solution is always stored internally, and is available before next optimization. The former optimal solution may be still feasible, but no longer optimal; or it may remain optimal if the modification of the objective function was small.

In general, **MOSEK** exploits dual information and availability of an optimal basis from the previous execution. The simplex optimizer is well suited for exploiting an existing primal or dual feasible solution. Restarting capabilities for interior-point methods are still not as reliable and effective as those for the simplex algorithm. More information can be found in Chapter 10 of the book *[Chv83]*.

Parameter settings (see Sec. 8.4) can also be changed between optimizations.

7.9.1 Example: Production Planning

A company manufactures three types of products. Suppose the stages of manufacturing can be split into three parts: Assembly, Polishing and Packing. In the table below we show the time required for each stage as well as the profit associated with each product.

Product no.	Assembly (minutes)	Polishing (minutes)	Packing (minutes)	Profit ($)
0	2	3	2	1.50
1	4	2	3	2.50
2	3	3	2	3.00

With the current resources available, the company has $100,000$ minutes of assembly time, $50,000$ minutes of polishing time and $60,000$ minutes of packing time available per year. We want to know how many items of each product the company should produce each year in order to maximize profit?

Denoting the number of items of each type by x_0, x_1 and x_2, this problem can be formulated as a linear optimization problem:

$$
\begin{array}{lccccccl}
\text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\
\text{subject to} & 2x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\
 & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\
 & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000,
\end{array}
\tag{7.18}
$$

and

$$x_0, x_1, x_2 \geq 0.$$

Code in Listing 7.33 loads and solves this problem.

Listing 7.33: Setting up and solving problem (7.18)

```
# Problem data
c = [ 1.5, 2.5, 3.0 ]
A = [ [2, 4, 3],
          [3, 2, 3],
          [2, 3, 2] ]
b = [ 100000.0, 50000.0, 60000.0 ]
numvar = len(c)
numcon = len(b)

# Create a model and input data
with Model() as M:
        x = M.variable("x",numvar, Domain.greaterThan(0.0))
        con = M.constraint(Expr.mul(A, x), Domain.lessThan(b))
        M.objective(ObjectiveSense.Maximize, Expr.dot(c, x))
        # Solve the problem
        M.solve()
```

7.9.2 Changing the Linear Constraint Matrix

Suppose we want to change the time required for assembly of product 0 to 3 minutes. This corresponds to setting $a_{0,0} = 3$. Now the *Constraint* provides the method *Constraint.update*, which can replace the columns corresponding to a variable with new values (or to replace the whole constraint). In our case the update we need is replacing $1 \cdot x_0$ with $3 \cdot x_0$ in the constraint with index 0.

```
        x0 = x.index(0)
        con.index(0).update(Expr.mul(3.0, x0), x0)
```

The problem now has the form:

$$
\begin{array}{lccccccl}
\text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\
\text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\
 & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\
 & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000,
\end{array} \tag{7.19}
$$

and

$$x_0, x_1, x_2 \geq 0.$$

After this operation we can reoptimize the problem.

7.9.3 Appending Variables

We now want to add a new product with the following data:

Product no.	Assembly (minutes)	Polishing (minutes)	Packing (minutes)	Profit ($)
3	4	0	1	1.00

This corresponds to creating a new variable x_3, appending a new column to the A matrix and setting a new term in the objective. We do this in Listing 7.34

Listing 7.34: How to add a new variable (column)

```
        ############## Add a new variable ################
        # Create a variable and a compound view of all variables
        x3 = M.variable("y",Domain.greaterThan(0.0))
        xNew = Var.vstack(x, x3)
        # Add to the exising constraint
        con.update(Expr.mul(x3, [4, 0, 1]), x3)
        # Change the objective to include x3
        M.objective(ObjectiveSense.Maximize, Expr.dot(c+[1.0], xNew))
```

After this operation the new problem is:

$$\begin{array}{lccccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 60000, \end{array} \tag{7.20}$$

and

$$x_0, x_1, x_2, x_3 \geq 0.$$

7.9.4 Appending Constraints

Now suppose we want to add a new stage to the production process called *Quality control* for which 30000 minutes are available. The time requirement for this stage is shown below:

Product no.	Quality control (minutes)
0	1
1	2
2	1
3	1

This corresponds to adding the constraint

$$x_0 + 2x_1 + x_2 + x_3 \leq 30000$$

to the problem. This is done as follows.

Listing 7.35: Adding a new constraint.

```
############## Add a new constraint ################
con2 = M.constraint(Expr.dot(xNew, [1, 2, 1, 1]), Domain.lessThan(30000.0))
```

Again, we can continue with re-optimizing the modified problem.

7.9.5 Changing bounds

One typical reoptimization scenario is to change bounds. Suppose for instance that we must operate with limited time resources, and we must change the upper bounds in the problem as follows:

Operation	Time available (before)	Time available (new)
Assembly	100000	80000
Polishing	50000	40000
Packing	60000	50000
Quality control	30000	22000

That means we would like to solve the problem:

$$\begin{array}{lccccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 80000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 40000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 50000, \\ & x_0 & + & 2x_1 & + & x_2 & + & x_3 & \leq & 22000. \end{array} \tag{7.21}$$

Since `Domain` objects are immutable, we cannot change the constraints by simply updating the value inside domains. To circumvent this, we add the differences between new and old bounds as fixed terms to the constraint expression. That means, we effectively construct an equivalent problem:

$$\begin{array}{lccccccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & + & 20000 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & + & 10000 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & + & 10000 & \leq & 60000, \\ & x_0 & + & 2x_1 & + & x_2 & + & x_3 & + & 8000 & \leq & 30000. \end{array} \tag{7.22}$$

The next listing shows how to do it.

Listing 7.36: Change constraint bounds.

```
############## Change constraint bounds ################
cAll = Constraint.vstack(con, con2)
# Change bounds by effectively updating fixed terms with the difference
cAll.update([20000, 10000, 10000, 8000])
```

Again, we can continue with re-optimizing the modified problem.

7.9.6 Advanced hot-start

If the optimizer used the data from the previous run to hot-start the optimizer for reoptimization, this will be indicated in the log:

```
Optimizer  - hotstart               : yes
```

When performing re-optimizations, instead of removing a basic variable it may be more efficient to fix the variable at zero and then remove it when the problem is re-optimized and it has left the basis. This makes it easier for **MOSEK** to restart the simplex optimizer.

For a more in-depth treatment see the following sections:

- *Case studies* for more advanced and complicated optimization examples.
- *Problem Formulation and Solutions* for formal mathematical formulations of problems **MOSEK** can solve, dual problems and infeasibility certificates.

Chapter 8

Solver Interaction Tutorials

In this section we cover the interaction with the solver.

8.1 Accessing the solution

This section contains important information about the status of the solver and the status of the solution, which must be checked in order to properly interpret the results of the optimization.

8.1.1 Solver termination

If an error occurs during optimization then the method *Model.solve* will throw an exception of type *OptimizeError*. The method *FusionRuntimeException.toString* will produce a description of the error, if available. More about exceptions in Sec. 8.2.

If a runtime error causes the program to crash during optimization, the first debugging step is to enable logging and check the log output. See Sec. 8.3.

If the optimization completes successfully, the next step is to check the solution status, as explained below.

8.1.2 Available solutions

MOSEK uses three kinds of optimizers and provides three types of solutions:

- **basic solution** from the simplex optimizer,
- **interior-point solution** from the interior-point optimizer,
- **integer solution** from the mixed-integer optimizer.

Under standard parameters settings the following solutions will be available for various problem types:

Table 8.1: Types of solutions available from **MOSEK**

	Simplex optimizer	Interior-point optimizer	Mixed-integer optimizer
Linear problem	*SolutionType.Basic*	*SolutionType.Interior*	
Conic (nonlinear) problem		*SolutionType.Interior*	
Problem with integer variables			*SolutionType.Integer*

For linear problems the user can force a specific optimizer choice making only one of the two solutions available. For example, if the user disables basis identification, then only the interior point solution will be available for a linear problem. Numerical issues may cause one of the solutions to be unknown even if another one is feasible.

Not all components of a solution are always available. For example, there is no dual solution for integer problems and no dual conic variables from the simplex optimizer.

The user will always need to specify which solution should be accessed.

Moreover, the user may be oblivious to the actual solution type by always referring to *SolutionType.Default*, which will automatically select the best available solution, if there is more than one. Moreover, the method *Model.selectedSolution* can be used to fix one solution type for all future references.

8.1.3 Problem and solution status

Assuming that the optimization terminated without errors, the next important step is to check the problem and solution status. There is one for every type of solution, as explained above.

Problem status

Problem status (*ProblemStatus*, retrieved with *Model.getProblemStatus*) determines whether the problem is certified as feasible. Its values can roughly be divided into the following broad categories:

- **feasible** — the problem is feasible. For continuous problems and when the solver is run with default parameters, the feasibility status should ideally be *ProblemStatus.PrimalAndDualFeasible*.
- **primal/dual infeasible** — the problem is infeasible or unbounded or a combination of those. The exact problem status will indicate the type of infeasibility.
- **unknown** — the solver was unable to reach a conclusion, most likely due to numerical issues.

Solution status

Solution status (*SolutionStatus*, retrieved with *Model.getPrimalSolutionStatus* and *Model.getDualSolutionStatus*) provides the information about what the solution values actually contain. The most important broad categories of values are:

- **optimal** (*SolutionStatus.Optimal*) — the solution values are feasible and optimal.
- **certificate** — the solution is in fact a certificate of infeasibility (primal or dual, depending on the solution).
- **unknown/undefined** — the solver could not solve the problem or this type of solution is not available for a given problem.

The solution status determines the action to be taken. For example, in some cases a suboptimal solution may still be valuable and deserve attention. It is the user's responsibility to check the status and quality of the solution.

Typical status reports

Here are the most typical optimization outcomes described in terms of the problem and solution statuses. Note that these do not cover all possible situations that can occur.

Table 8.2: Continuous problems (solution status for `SolutionType.Interior` or `SolutionType.Basic`)

Outcome	Problem status	Solution status (primal)	Solution status (dual)
Optimal	*ProblemStatus.PrimalAndDualFeasible*	*SolutionStatus.Optimal*	*SolutionStatus.Optimal*
Primal infeasible	*ProblemStatus.PrimalInfeasible*	*SolutionStatus.Unknown*	*SolutionStatus.Certificate*
Dual infeasible (unbounded)	*ProblemStatus.DualInfeasible*	*SolutionStatus.Certificate*	*SolutionStatus.Unknown*
Uncertain (stall, numerical issues, etc.)	*ProblemStatus.Unknown*	*SolutionStatus.Unknown*	*SolutionStatus.Unknown*

Table 8.3: Integer problems (solution status for `SolutionType.Integer`, others undefined)

Outcome	Problem status	Solution status (primal)	Solution status (dual)
Integer optimal	*ProblemStatus.PrimalFeasible*	*SolutionStatus.Optimal*	*SolutionStatus.Unknown*
Infeasible	*ProblemStatus.PrimalInfeasible*	*SolutionStatus.Unknown*	*SolutionStatus.Unknown*
Integer feasible point	*ProblemStatus.PrimalFeasible*	*SolutionStatus.Feasible*	*SolutionStatus.Unknown*
No conclusion	*ProblemStatus.Unknown*	*SolutionStatus.Unknown*	*SolutionStatus.Unknown*

8.1.4 Retrieving solution values

After the meaning and quality of the solution (or certificate) have been established, we can query for the actual numerical values. They can be accessed using:

- *Model.primalObjValue*, *Model.dualObjValue* — the primal and dual objective value.
- *Variable.level* — solution values for the variables.
- *Constraint.level* — values of the constraint expressions in the current solution.
- *Constraint.dual*, *Variable.dual* — dual values.

Remark

By default only *optimal solutions* are returned. An attempt to access a solution with a weaker status will result in an exception. This can be changed by choosing another level of *acceptable* solutions with the method *Model.acceptedSolutionStatus*. In particular, this method must be called to enable retrieving suboptimal solutions and infeasibility certificates. For instance, one could write

```
  M.acceptedSolutionStatus(AccSolutionStatus.Feasible)
```

The current setting of acceptable solutions can be checked with *Model.getAcceptedSolutionStatus*.

8.1.5 Source code example

Below is a source code example with a simple framework for assessing and retrieving the solution to a conic optimization problem.

Listing 8.1: Sample framework for checking optimization result.

```
with Model() as M:
  # (Optional) set a log stream
  # M.setLoghandler(sys.stdout)

  # (Optional) uncomment to see what happens when solution status is unknown
  # M.setSolverParam("intpntMaxIterations", 1)

  # In this example we set up a small conic problem
  setupExample(M)

  # Optimize
  try:
    M.solve()

    # We expect solution status OPTIMAL (this is also default)
    M.acceptedSolutionStatus(AccSolutionStatus.Optimal)
```

(continues on next page)

(continued from previous page)

```
    print("Optimal solution for x: {0}".format(M.getVariable('x').level()))
    print("Optimal primal objective: {0}".format(M.primalObjValue()))
    # .. continue analyzing the solution

  except OptimizeError as e:
    print("Optimization failed. Error: {0}".format(e))

  except SolutionError as e:
    # The solution with at least the expected status was not available.
    # We try to diagnoze why.
    print("Requested solution was not available.")
    prosta = M.getProblemStatus()

    if prosta == ProblemStatus.DualInfeasible:
      print("Dual infeasibility certificate found.")

    elif prosta == ProblemStatus.PrimalInfeasible:
      print("Primal infeasibility certificate found.")

    elif prosta == ProblemStatus.Unknown:
      # The solutions status is unknown. The termination code
      # indicates why the optimizer terminated prematurely.
      print("The solution status is unknown.")
      symname, desc = mosek.Env.getcodedesc(mosek.rescode(int(M.getSolverIntInfo(
↪"optimizeResponse"))))
      print("   Termination code: {0} {1}".format(symname, desc))

    else:
      print("Another unexpected problem status {0} is obtained.".format(prosta))

  except Exception as e:
    print("Unexpected error: {0}".format(e))
```

8.2 Errors and exceptions

Exceptions

Almost every method in Fusion API for Python can throw an exception informing that the requested operation was not performed correctly, and indicating the type of error that occurred. This is the case in situations such as for instance:

- incompatible dimensions in a linear expression,
- defining an invalid value for a parameter,
- accessing an undefined solution,
- repeating a variable name, etc.

It is therefore a good idea to catch exceptions of type `FusionException` and its specific subclasses. The one case where it is *extremely important* to do so is when `Model.solve` is invoked. We will say more about this in Sec. 8.1.

The exception contains a short diagnostic message. They can be accessed as in the following example.

```
with Model() as M:
    try:
        M.setSolverParam("intpntCoTolRelGap", 1.01)
    except mosek.fusion.ParameterError as e:
        print("Error: {0}".format(e))
```

It will produce as output:

```
Error: Invalid value for parameter (intpntCoTolRelGap)
```

Optimizer errors and warnings

The optimizer may also produce warning messages. They indicate non-critical but important events, that will not prevent solver execution, but may be an indication that something in the optimization problem might be improved. Warning messages are normally printed to a log stream (see Sec. 8.3). A typical warning is, for example:

```
MOSEK warning 53: A numerically large upper bound value  6.6e+09 is specified for constraint
↪'C69200' (46020).
```

Error and solution status handling example

Below is a source code example with a simple framework for handling major errors when assessing and retrieving the solution to a conic optimization problem.

Listing 8.2: Sample framework for checking optimization result.

```
with Model() as M:
  # (Optional) set a log stream
  # M.setLoghandler(sys.stdout)

  # (Optional) uncomment to see what happens when solution status is unknown
  # M.setSolverParam("intpntMaxIterations", 1)

  # In this example we set up a small conic problem
  setupExample(M)

  # Optimize
  try:
    M.solve()

    # We expect solution status OPTIMAL (this is also default)
    M.acceptedSolutionStatus(AccSolutionStatus.Optimal)

    print("Optimal solution for x: {0}".format(M.getVariable('x').level()))
    print("Optimal primal objective: {0}".format(M.primalObjValue()))
    # .. continue analyzing the solution

  except OptimizeError as e:
    print("Optimization failed. Error: {0}".format(e))

  except SolutionError as e:
    # The solution with at least the expected status was not available.
    # We try to diagnoze why.
    print("Requested solution was not available.")
    prosta = M.getProblemStatus()

    if prosta == ProblemStatus.DualInfeasible:
      print("Dual infeasibility certificate found.")

    elif prosta == ProblemStatus.PrimalInfeasible:
      print("Primal infeasibility certificate found.")

    elif prosta == ProblemStatus.Unknown:
      # The solutions status is unknown. The termination code
      # indicates why the optimizer terminated prematurely.
      print("The solution status is unknown.")
      symname, desc = mosek.Env.getcodedesc(mosek.rescode(int(M.getSolverIntInfo(
↪"optimizeResponse"))))
```

(continues on next page)

(continued from previous page)

```
      print("   Termination code: {0} {1}".format(symname, desc))

    else:
      print("Another unexpected problem status {0} is obtained.".format(prosta))

  except Exception as e:
    print("Unexpected error: {0}".format(e))
```

8.3 Input/Output

The *Model* class is also a proxy for input/output operations related to an optimization model.

8.3.1 Stream logging

By default the solver runs silently and does not produce any output to the console or otherwise. However, the log output can be redirected to a user-defined output stream or stream callback function. The log output is analogous to the one produced by the command-line version of **MOSEK**.

To redirect all log messages use the method *Model.setLogHandler*. For instance, we can use the standard output:

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

A log stream can be detached by passing NULL.

8.3.2 Log verbosity

The logging verbosity can be controlled by setting the relevant parameters, as for instance

- *log*,
- *logIntpnt*,
- *logMio*,
- *logCutSecondOpt*,
- *logSim*, and
- *logSimMinor*.

Each parameter controls the output level of a specific functionality or algorithm. The main switch is *log* which affect the whole output. The actual log level for a specific functionality is determined as the minimum between *log* and the relevant parameter. For instance, the log level for the output produce by the interior-point algorithm is tuned by the *logIntpnt*; the actual log level is defined by the minimum between *log* and *logIntpnt*.

Tuning the solver verbosity may require adjusting several parameters. It must be noticed that verbose logging is supposed to be of interest during debugging and tuning. When output is no more of interest, the user can easily disable it globally with *log*. Larger values of *log* do not necessarily result in increased output.

By default **MOSEK** will reduce the amount of log information after the first optimization on a given problem. To get full log output on subsequent re-optimizations set *logCutSecondOpt* to zero.

8.3.3 Saving a problem to a file

An optimization model defined in *Fusion* can be dumped to a file using the method *Model.writeTask*. The file format will be determined from the filename's extension. Supported formats are listed in Sec. 15 together with a table of problem types supported by each.

For instance the problem can be written to an OPF file with

```
M.writeTask('dump.opf')
```

All formats can be compressed with `gzip` by appending the `.gz` extension, for example

```
M.writeTask('dump.task.gz')
```

Some remarks:

- The problem is written to the file as it is represented in the underlying *optimizer task*, that is including auxiliary variables introduced by *Fusion* if necessary.
- Unnamed variables are given generic names. It is therefore recommended to use meaningful variable names if the problem file is meant to be human-readable.
- The `task` format is **MOSEK**'s native file format which contains all the problem data as well as solver settings.

8.3.4 Reading a problem from a file

It is not possible to read a file saved with *Model.writeTask* back into *Fusion* because the structure of the high-level optimization model is not saved. However, such problem files can be solved with the command-line tool or read by the low-level Optimizer API if necessary. See the documentation of those interfaces for details.

8.4 Setting solver parameters

MOSEK comes with a large number of parameters that allows the user to tune the behavior of the optimizer. The typical settings which can be changed with solver parameters include:

- choice of the optimizer for linear problems,
- choice of primal/dual solver,
- turning presolve on/off,
- turning heuristics in the mixed-integer optimizer on/off,
- level of multi-threading,
- feasibility tolerances,
- solver termination criteria,
- behaviour of the license manager,

and more. All parameters have default settings which will be suitable for most typical users.
The API reference contains:

- *Full list of parameters*
- *List of parameters grouped by topic*

Setting parameters

Each parameter is identified by a unique string name and it can accept either integers, floating point values or symbolic strings. Parameters are set using the method *Model.setSolverParam*. *Fusion* will try to convert the given argument to the exact expected type, and will raise an exception if that fails.

Some parameters accept only symbolic strings from a fixed set of values. The set of accepted values for every parameter is provided in the API reference.

For example, the following piece of code sets up parameters which choose and tune the interior point optimizer before solving a problem.

Listing 8.3: Parameter setting example.

```
# Set log level (integer parameter)
M.setSolverParam("log", 1)
# Select interior-point optimizer... (parameter with symbolic string values)
M.setSolverParam("optimizer", "intpnt")
# ... without basis identification (parameter with symbolic string values)
M.setSolverParam("intpntBasis", "never")
# Set relative gap tolerance (double parameter)
M.setSolverParam("intpntCoTolRelGap", 1.0e-7)

# The same in a different way
M.setSolverParam("intpntCoTolRelGap", "1.0e-7")

# Incorrect value
try:
    M.setSolverParam("intpntCoTolRelGap", -1)
except ParameterError as e:
    print('Wrong parameter value')
```

8.5 Retrieving information items

After the optimization the user has access to the solution as well as to a report containing a large amount of additional *information items*. For example, one can obtain information about:

- **timing**: total optimization time, time spent in various optimizer subroutines, number of iterations, etc.
- **solution quality**: feasibility measures, solution norms, constraint and bound violations, etc.
- **problem structure**: counts of variables of different types, constraints, nonzeros, etc.
- **integer optimizer**: integrality gap, objective bound, number of cuts, etc.

and more. Information items are numerical values of integer, long integer or double type. The full list can be found in the API reference:

- *Double information items*
- *Integer information items*
- *Long information items*

Certain information items make sense, and are made available, also *during* the optimization process. They can be accessed from a callback function, see Sec. 8.7 for details.

Remark

For efficiency reasons, not all information items are automatically computed after optimization. To force all information items to be updated use the parameter *autoUpdateSolInfo*.

Retrieving the values

Values of information items are fetched using one of the methods

- *Model.getSolverDoubleInfo* for a double information item,
- *Model.getSolverIntInfo* for an integer information item,
- *Model.getSolverLIntInfo* for a long integer information item.

Each information item is identified by a unique name. The example below reads two pieces of data from the solver: total optimization time and the number of interior-point iterations.

Listing 8.4: Information items example.

```
tm = M.getSolverDoubleInfo("optimizerTime")
it = M.getSolverIntInfo("intpntIter")

print('Time: {0}\nIterations: {1}'.format(tm,it))
```

8.6 Stopping the solver

The *Model* provides the method *Model.breakSolver* that notifies the solver that it must stop as soon as possible. The solver will not terminate momentarily, as it only periodically checks for such notifications. In any case, it will stop as soon as possible. The typical usage pattern of this method would be:

- build the optimization model M,
- create a separate thread in which M will run,
- break the solver by calling *Model.breakSolver* from the main thread.

Warnings and comments:

- It is recommended to use the solver parameters to set or modify standard built-in termination criteria (such as maximal running time, solution tolerances etc.). See Sec. 8.4.
- More complicated user-defined termination criteria can be implemented within a callback function. See Sec. 8.7.
- The state of the solver and solution after termination may be undefined.
- This operation is very language dependent and particular care must be taken to avoid stalling or other undesired side effects.

8.6.1 Example: Setting a Time Limit

For the purpose of the tutorial we will implement a busy-waiting breaker with the time limit as a termination criterion. Note that in practice it would be better just to set the parameter *optimizerMaxTime*.

Suppose we built a model M that is known to run for quite a long time (in the accompanying example code we create a particular integer program). Then we could create a new thread solving the model:

```
        T = threading.Thread(target=M.solve)
```

In the main thread we are going to check if one of the two criteria are satisfied:

- a time limit has elapsed,
- the user pressed CTRL+C.

After calling *Model.breakSolver* we should wait for the solver thread to actually return. Altogether this scenario can be implemented as follows:

Listing 8.5: Stopping solver execution.

```
        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():
```

(continues on next page)

(continued from previous page)

```
                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
```

8.7 Progress and data callback

Callbacks are a very useful mechanism that allow the caller to track the progress of the **MOSEK** optimizer. A callback function provided by the user is regularly called during the optimization and can be used to

- obtain a customized log of the solver execution,
- collect information for debugging purposes or
- ask the solver to terminate.

Fusion API for Python has the following callback mechanisms:

- **progress callback**, which provides only the basic status of the solver.
- **data callback**, which provides the solver status and a complete set of information items that describe the progress of the optimizer in detail.

Warning

The callbacks functions *must not* invoke any functions of the solver, environment or task. Otherwise the state of the solver and its outcome are undefined.

8.7.1 Data callback

In the data callback **MOSEK** passes a callback code and values of all information items to a user-defined function. The callback function is called, in particular, at the beginning of each iteration of the interior-point optimizer. For the simplex optimizers *logSimFreq* controls how frequently the call-back is called. Note that the callback is done quite frequently, which can lead to degraded performance. If the information items are not required, the simpler progress callback may be a better choice.

The data callback is set by calling the method *Model.setDataCallbackHandler*.

The callback function should have the following signature:

```
def userCallback(caller,
                 douinf,
                 intinf,
                 lintinf):
```

Arguments:

- `caller` - the status of the optimizer.
- `douinf` - values of double information items.
- `intinf` - values of integer information items.

- `lintinf` - values of long information items.

Return value: Non-zero return value of the callback function indicates that the optimizer should be terminated.

8.7.2 Progress callback

In the progress callback **MOSEK** provides a single code indicating the current stage of the optimization process.

The callback is set by calling the method *Model.setCallbackHandler*.

The callback function should have the following signature

```
def userProgresCallback(caller):
```

Arguments:

- `caller` - the status of the optimizer.

Return value: Non-zero return value of the callback function indicates that the optimizer should be terminated.

8.7.3 Working example: Data callback

The following example defines a data callback function that prints out some of the information items. It interrupts the solver after a certain time limit. Note that the time limit refers to time spent in the solver and does not include setting up the model in *Fusion*.

Listing 8.6: An example of a data callback function.

```
def makeUserCallback(model, 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.")
```

(continues on next page)

(continued from previous page)

```
        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.
            print("Too much time, terminating.")
            return 1
        return 0

    return userCallback
```

Assuming that we have defined a model `M` and a time limit `maxtime`, the callback function is attached as follows:

Listing 8.7: Attaching the data callback function to the model.

```
        userCallback = makeUserCallback(model=M, maxtime=0.07)
        M.setDataCallbackHandler(userCallback)
```

8.8 Optimizer API Task

This section is intended for advanced users and should normally never be followed unless advanced debugging or very specialized functionalities are required.

The *Model* is a wrapper on top of an underlying **MOSEK** low-level optimizer task. Access to the task is provided by the method *Model.getTask*. The functionalities available from the task are described in the documentation of the relevant Optimizer API.

Warning

Note that the user gets access to the *actual task* in the model, and *not* its clone. Changing the state of the task will most likely invalidate the *Fusion* model.

Chapter 9

Debugging Tutorials

This collection of tutorials contains basic techniques for debugging optimization problems using tools available in **MOSEK**: optimizer log, solution summary, infeasibility report, command-line tools. It is intended as a first line of technical help for issues such as: Why do I get solution status *unknown* and how can I fix it? Why is my model infeasible while it shouldn't be? Should I change some parameters? Can the model solve faster? etc.

The major steps when debugging a model are always:

- Enable log output. See Sec. 8.3.1 for how to do it. In the simplest case:

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

- Run the optimization and analyze the log output, see Sec. 9.1. In particular:
  - check if the problem setup (number of constraints/variables etc.) matches your expectation.
  - check solution summary and solution status.
- Dump the problem to disk if necessary to continue analysis. See Sec. 8.3.3.
  - use a human-readable text format, such as `*.opf` if you want to check the problem structure by hand. Assign names to variables and constraints to make them easier to identify.

    ```
    M.writeTask('dump.opf')
    ```

  - use the **MOSEK** native format `*.task.gz` when submitting a bug report or support question.

    ```
    M.writeTask('dump.task.gz')
    ```

- Fix problem setup, improve the model, locate infeasibility or adjust parameters, depending on the diagnosis.

See the following sections for details.

9.1 Understanding optimizer log

The optimizer produces a log which splits roughly into four sections:

1. summary of the input data,
2. presolve and other pre-optimize problem setup stages,
3. actual optimizer iterations,
4. solution summary.

In this tutorial we show how to analyze the most important parts of the log when initially debugging a model: input data (1) and solution summary (4). For the iterations log (3) see Sec. 13.3.4 or Sec. 13.4.8.

9.1.1 Input data

If **MOSEK** behaves very far from expectations it may be due to errors in problem setup. The log file will begin with a summary of the structure of the problem, which looks for instance like:

```
Problem
  Name                   :
  Objective sense        : max
  Type                   : CONIC (conic optimization problem)
  Constraints            : 20413
  Cones                  : 2508
  Scalar variables       : 20414
  Matrix variables       : 0
  Integer variables      : 0
```

This can be consulted to eliminate simple errors: wrong objective sense, wrong number of variables etc. Note that Fusion, and third-party modeling tools can introduce additional variables and constraints to the model. In the remaining **MOSEK** APIs the problem dimensions should match exactly what the user specified.

If this is not sufficient a bit more information can be obtained by dumping the problem to a file (see Sec. 9) and using the `anapro` option of any of the command line tools. This will produce a longer summary similar to:

```
** Variables
scalar: 20414     integer: 0        matrix: 0
low: 2082         up: 5014          ranged: 0         free: 12892       fixed: 426

** Constraints
all: 20413
low: 10028        up: 0             ranged: 0         free: 0           fixed: 10385

** Cones
QUAD: 1           dims: 2865: 1
RQUAD: 2507       dims: 3: 2507

** Problem data (numerics)
|c|               nnz: 10028        min=2.09e-05      max=1.00e+00
|A|               nnz: 597023       min=1.17e-10      max=1.00e+00
blx               fin: 2508         min=-3.60e+09     max=2.75e+05
bux               fin: 5440         min=0.00e+00      max=2.94e+08
blc               fin: 20413        min=-7.61e+05     max=7.61e+05
buc               fin: 10385        min=-5.00e-01     max=0.00e+00
```

Again, this can be used to detect simple errors, such as:

- Wrong type of cone was used or it has wrong dimension.
- The bounds for variables or constraints are incorrect or incomplete.
- The model is otherwise incomplete.
- Suspicious values of coefficients.
- For various data sizes the model does not scale as expected.

Finally saving the problem in a human-friendly text format such as LP or OPF (see Sec. 9) and analyzing it by hand can reveal if the model is correct.

Warnings and errors

At this stage the user can encounter warnings which should not be ignored, unless they are well-understood. They can also serve as hints as to numerical issues with the problem data. A typical warning of this kind is

```
MOSEK warning 53: A numerically large upper bound value  2.9e+08 is specified for variable
↪'absh[107]' (2613).
```

Warnings do not stop the problem setup. If, on the other hand, an error occurs then the model will become invalid. The user should make sure to test for errors/exceptions from all API calls that set up the problem and validate the data. See Sec. 8.2 for more details.

9.1.2 Solution summary

The last item in the log is the solution summary.

Continuous problem

Optimal solution

A typical solution summary for a continuous (linear, conic, quadratic) problem looks like:

```
Problem status  : PRIMAL_AND_DUAL_FEASIBLE
Solution status : OPTIMAL
Primal.  obj: 8.7560516107e+01    nrm: 1e+02    Viol.  con: 3e-12    var: 0e+00    cones: 3e-11
Dual.    obj: 8.7560521345e+01    nrm: 1e+00    Viol.  con: 5e-09    var: 9e-11    cones: 0e+00
```

It contains the following elements:

- Problem and solution status. For details see Sec. 8.1.3.
- A summary of the primal solution: objective value, infinity norm of the solution vector `xx`, maximal violations of constraints, variable bounds and cones. The violation of a linear constraint such as $a^T x \leq b$ is $\max(a^T x - b, 0)$. The violation of a conic constraint $x \in \mathcal{K}$ is the distance $\mathrm{dist}(x, \mathcal{K})$.
- The same for the dual solution.

The features of the solution summary which characterize a very good and accurate solution and a well-posed model are:

- **Status:** The solution status is `OPTIMAL`.
- **Duality gap:** The primal and dual objective values are (almost) identical, which proves the solution is (almost) optimal.
- **Norms:** Ideally the norms of the solution and the objective values should not be too large. This of course depends on the input data, but a huge solution norm can be an indicator of issues with the scaling, conditioning and/or well-posedness of the model. It may also indicate that the problem is borderline between feasibility and infeasibility and sensitive to small perturbations in this respect.
- **Violations:** The violations are close to zero, which proves the solution is (almost) feasible. Observe that due to rounding errors it can be expected that the violations are proportional to the norm (`nrm:`) of the solution. It is rarely the case that violations are exactly zero.

Solution status UNKNOWN

A typical example with solution status `UNKNOWN` due to numerical problems will look like:

```
Problem status  : UNKNOWN
Solution status : UNKNOWN
Primal.  obj: 1.3821656824e+01    nrm: 1e+01    Viol.  con: 2e-03    var: 0e+00    cones: 0e+00
Dual.    obj: 3.0119004098e-01    nrm: 5e+07    Viol.  con: 4e-16    var: 1e-01    cones: 0e+00
```

Note that:

- The primal and dual objective are very different.
- The dual solution has very large norm.
- There are considerable violations so the solution is likely far from feasible.

Follow the hints in Sec. 9.2 to resolve the issue.

Solution status UNKNOWN with a potentially useful solution

Solution status UNKNOWN does not necessarily mean that the solution is completely useless. It only means that the solver was unable to make any more progress due to numerical difficulties, and it was not able to reach the accuracy required by the termination criteria (see Sec. 13.3.2). Consider for instance:

```
Problem status  : UNKNOWN
Solution status : UNKNOWN
Primal.  obj: 3.4531019648e+04    nrm: 1e+05    Viol.  con: 7e-02    var: 0e+00    cones: 0e+00
Dual.    obj: 3.4529720645e+04    nrm: 8e+03    Viol.  con: 1e-04    var: 2e-04    cones: 0e+00
```

Such a solution may still be useful, and it is always up to the user to decide. It may be a good enough approximation of the optimal point. For example, the large constraint violation may be due to the fact that one constraint contained a huge coefficient.

Infeasibility certificate

A primal infeasibility certificate is stored in the dual variables:

```
Problem status  : PRIMAL_INFEASIBLE
Solution status : PRIMAL_INFEASIBLE_CER
Dual.    obj: 2.9238975853e+02    nrm: 6e+02    Viol.  con: 0e+00    var: 1e-11    cones: 0e+00
```

It is a Farkas-type certificate as described in Sec. 12.2.2. In particular, for a good certificate:

- The dual objective is positive for a minimization problem, negative for a maximization problem. Ideally it is well bounded away from zero.
- The norm is not too big and the violations are small (as for a solution).

If the model was not expected to be infeasible, the likely cause is an error in the problem formulation. Use the hints in Sec. 9.1.1 and Sec. 9.3 to locate the issue.

Just like a solution, the infeasibility certificate can be of better or worse quality. The infeasibility certificate above is very solid. However, there can be less clear-cut cases, such as for example:

```
Problem status  : PRIMAL_INFEASIBLE
Solution status : PRIMAL_INFEASIBLE_CER
Dual.    obj: 1.6378689238e-06    nrm: 6e+05    Viol.  con: 7e-03    var: 2e-04    cones: 0e+00
```

This infeasibility certificate is more dubious because the dual objective is positive, but barely so in comparison with the large violations. It also has rather large norm. This is more likely an indication that the problem is borderline between feasibility and infeasibility or simply ill-posed and sensitive to tiny variations in input data. See Sec. 9.3 and Sec. 9.2.

The same remarks apply to dual infeasibility (i.e. unboundedness) certificates. Here the primal objective should be negative a minimization problem and positive for a maximization problem.

9.1.3 Mixed-integer problem

Optimal integer solution

For a mixed-integer problem there is no dual solution and a typical optimal solution report will look as follows:

```
Problem status  : PRIMAL_FEASIBLE
Solution status : INTEGER_OPTIMAL
Primal.  obj: 6.0111122960e+06    nrm: 1e+03    Viol.  con: 2e-13    var: 2e-14    itg: 5e-15
```

The interpretation of all elements is as for a continuous problem. The additional field `itg` denotes the maximum violation of an integer variable from being an exact integer.

Feasible integer solution

If the solver found an integer solution but did not prove optimality, for instance because of a time limit, the solution status will be PRIMAL_FEASIBLE:

```
Problem status  : PRIMAL_FEASIBLE
Solution status : PRIMAL_FEASIBLE
Primal.  obj: 6.0114607792e+06    nrm: 1e+03    Viol.  con: 2e-13    var: 2e-13    itg: 4e-15
```

In this case it is valuable to go back to the optimizer summary to see how good the best solution is:

```
31       35       1        0        6.0114607792e+06     6.0078960892e+06     0.06        4.1

Objective of best integer solution : 6.011460779193e+06
Best objective bound               : 6.007896089225e+06
```

In this case the best integer solution found has objective value `6.011460779193e+06`, the best proved lower bound is `6.007896089225e+06` and so the solution is guaranteed to be within 0.06% from optimum. The same data can be obtained as information items through an API. See also Sec. 13.4 for more details.

Infeasible problem

If the problem is declared infeasible the summary is simply

```
Problem status  : PRIMAL_INFEASIBLE
Solution status : UNKNOWN
Primal.  obj: 0.0000000000e+00    nrm: 0e+00    Viol.  con: 0e+00    var: 0e+00    itg: 0e+00
```

If infeasibility was not expected, consult Sec. 9.3.

9.2 Addressing numerical issues

The suggestions in this section should help diagnose and solve issues with numerical instability, in particular `UNKNOWN` solution status or solutions with large violations. Since numerically stable models tend to solve faster, following these hints can also dramatically shorten solution times.

We always recommend that issues of this kind are addressed by reformulating or rescaling the model, since it is the modeler who has the best insight into the structure of the problem and can fix the cause of the issue.

9.2.1 Formulating problems

Scaling

Make sure that all the data in the problem are of comparable orders of magnitude. This applies especially to the linear constraint matrix. Use Sec. 9.1.1 if necessary. For example a report such as

```
|A|               nnz: 597023       min=1.17e-6      max=2.21e+5
```

means that the ratio of largest to smallest elements in `A` is 10^{11}. In this case the user should rescale or reformulate the model to avoid such spread which makes it difficult for **MOSEK** to scale the problem internally. In many cases it may be possible to change the units, i.e. express the model in terms of rescaled variables (for instance work with millions of dollars instead of dollars, etc.).

Similarly, if the objective contains very different coefficients, say

$$\text{maximize } 10^{10}x + y$$

then it is likely to lead to inaccuracies. The objective will be dominated by the contribution from x and y will become insignificant.

Removing huge bounds

Never use a very large number as replacement for ∞. Instead define the variable or constraint as unbounded from below/above. Similarly, avoid artificial huge bounds if you expect they will not become tight in the optimal solution.

Avoiding linear dependencies

As much as possible try to avoid linear dependencies and near-linear dependencies in the model. See Example 9.3.

Avoiding ill-posedness

Avoid continuous models which are ill-posed: the solution space is degenerate, for example consists of a single point (technically, the Slater condition is not satisfied). In general, this refers to problems which are borderline between feasible and infeasible. See Example 9.1.

Scaling the expected solution

Try to formulate the problem in such a way that the expected solution (both primal and dual) is not very large. Consult the solution summary Sec. 9.1.2 to check the objective values or solution norms.

9.2.2 Further suggestions

Here are other simple suggestions that can help locate the cause of the issues. They can also be used as hints for how to tune the optimizer if fixing the root causes of the issue is not possible.

- Remove the objective and solve the feasibility problem. This can reveal issues with the objective.
- Change the objective or change the objective sense from minimization to maximization (if applicable). If the two objective values are almost identical, this may indicate that the feasible set is very small, possibly degenerate.
- Perturb the data, for instance bounds, very slightly, and compare the results.
- For linear problems: solve the problem using a different optimizer by setting the parameter *optimizer* and compare the results.
- Force the optimizer to solve the primal/dual versions of the problem by setting the parameter *intpntSolveForm* or *simSolveForm*. **MOSEK** has a heuristic to decide whether to dualize, but for some problems the guess is wrong an explicit choice may give better results.
- Solve the problem without presolve or some of its parts by setting the parameter *presolveUse*, see Sec. 13.1.
- Use different numbers of threads (*numThreads*) and compare the results. Very different results indicate numerical issues resulting from round-off errors.

If the problem was dumped to a file, experimenting with various parameters is facilitated with the **MOSEK** Command Line Tool or **MOSEK** Python Console Sec. 9.4.

9.2.3 Typical pitfalls

Example 9.1 (Ill-posedness)**.** A toy example of this situation is the feasibility problem

$$(x-1)^2 \leq 1, \ (x+1)^2 \leq 1$$

whose only solution is $x = 0$ and moreover replacing any 1 on the right hand side by $1-\varepsilon$ makes the problem infeasible and replacing it by $1+\varepsilon$ yields a problem whose solution set is an interval (fully-dimensional). This is an example of ill-posedness.

Example 9.2 (Huge solution)**.** If the norm of the expected solution is very large it may lead to numerical issues or infeasibility. For example the problem

$$(10^{-4}, x, 10^3) \in \mathcal{Q}_r^3$$

may be declared infeasible because the expected solution must satisfy $x \geq 5 \cdot 10^9$.

Example 9.3 (Near linear dependency). Consider the following problem:

$$
\begin{array}{lrrrrrrrrcl}
\text{minimize} & & & & & & & & & & \\
\text{subject to} & & x_1 & + & x_2 & & & & & = & 1, \\
& & & & & & x_3 & + & x_4 & = & 1, \\
& - & x_1 & & & - & x_3 & & & = & -1+\varepsilon, \\
& & & - & x_2 & & & - & x_4 & = & -1, \\
& & x_1, & & x_2, & & x_3, & & x_4 & \geq & 0.
\end{array}
$$

If we add the equalities together we obtain:

$$0 = \varepsilon$$

which is infeasible for any $\varepsilon \neq 0$. Here infeasibility is caused by a linear dependency in the constraint matrix coupled with a precision error represented by the ε. Indeed if a problem contains linear dependencies then the problem is either infeasible or contains redundant constraints. In the above case any of the equality constraints can be removed while not changing the set of feasible solutions. To summarize linear dependencies in the constraints can give rise to infeasible problems and therefore it is better to avoid them.

Example 9.4 (Presolving very tight bounds). Next consider the problem

$$
\begin{array}{lrcl}
\text{minimize} & & & \\
\text{subject to} & x_1 - 0.01x_2 & = & 0, \\
& x_2 - 0.01x_3 & = & 0, \\
& x_3 - 0.01x_4 & = & 0, \\
& x_1 & \geq & -10^{-9}, \\
& x_1 & \leq & 10^{-9}, \\
& x_4 & \geq & 10^{-4}.
\end{array}
$$

Now the **MOSEK** presolve will, for the sake of efficiency, fix variables (and constraints) that have tight bounds where tightness is controlled by the parameter *presolveTolX*. Since the bounds

$$-10^{-9} \leq x_1 \leq 10^{-9}$$

are tight, presolve will set $x_1 = 0$. It easy to see that this implies $x_4 = 0$, which leads to the incorrect conclusion that the problem is infeasible. However a tiny change of the value 10^{-9} makes the problem feasible. In general it is recommended to avoid ill-posed problems, but if that is not possible then one solution is to reduce parameters such as *presolveTolX* to say 10^{-10}. This will at least make sure that presolve does not make the undesired reduction.

9.3 Debugging infeasibility

This section contains hints for debugging problems that are unexpectedly infeasible. It is always a good idea to remove the objective, i.e. only solve a feasibility problem when debugging such issues.

9.3.1 Numerical issues

Infeasible problem status may be just an artifact of numerical issues appearing when the problem is badly-scaled, barely feasible or otherwise ill-conditioned so that it is unstable under small perturbations of the data or round-off errors. This may be visible in the solution summary if the infeasibility certificate has poor quality. See Sec. 9.1.2 for how to diagnose that and Sec. 9.2 for possible hints. Sec. 9.2.3 contains examples of situations which may lead to infeasibility for numerical reasons.

We refer to Sec. 9.2 for further information on dealing with those sort of issues. For the rest of this section we concentrate on the case when the solution summary leaves little doubt that the problem solved by the optimizer actually is infeasible.

9.3.2 Locating primal infeasibility

As an example of a primal infeasible problem consider minimizing the cost of transportation between a number of production plants and stores: Each plant produces a fixed number of goods, and each store has a fixed demand that must be met. Supply, demand and cost of transportation per unit are given in Fig. 9.1.

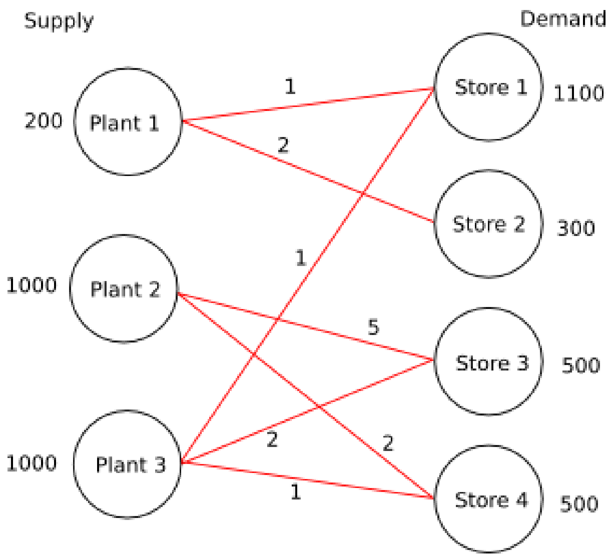


Fig. 9.1: Supply, demand and cost of transportation.

The problem represented in Fig. 9.1 is infeasible, since the total demand

$$2300 = 1100 + 200 + 500 + 500$$

exceeds the total supply

$$2200 = 200 + 1000 + 1000$$

If we denote the number of transported goods from plant i to store j by x_{ij}, the problem can be formulated as the LP:

$$\begin{array}{llcccccccccccccccl}
\text{minimize} & & x_{11} & + & 2x_{12} & + & 5x_{23} & + & 2x_{24} & + & x_{31} & + & 2x_{33} & + & x_{34} & & \\
\text{subject to} & s_0: & x_{11} & + & x_{12} & & & & & & & & & & & \leq & 200, \\
& s_1: & & & & & x_{23} & + & x_{24} & & & & & & & \leq & 1000, \\
& s_2: & & & & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\
& d_1: & x_{11} & & & & & & & + & x_{31} & & & & & = & 1100, \\
& d_2: & & & x_{12} & & & & & & & & & & & = & 200, \\
& d_3: & & & & & x_{23} & + & & & & & x_{33} & & & = & 500, \\
& d_4: & & & & & & & x_{24} & + & & & & & x_{34} & = & 500, \\
& & & & & & & & & & & & & & x_{ij} & \geq & 0.
\end{array} \tag{9.1}$$

Solving problem (9.1) using **MOSEK** will result in an infeasibility status. The infeasibility certificate is contained in the dual variables an can be accessed from an API. The variables and constraints with nonzero solution values form an infeasible subproblem, which frequently is very small. See Sec. 12.1.2 or Sec. 12.2.2 for detailed specifications of infeasibility certificates.

A short infeasibility report can also be printed to the log stream. It can be turned on by setting the parameter `MSK_IPAR_INFEAS_REPORT_AUTO` to `MSK_ON` in the command-line tool. This causes **MOSEK** to print a report on variables and constraints which are involved in infeasibility in the above sense, i.e. have nonzero values in the certificate. The parameter `MSK_IPAR_INFEAS_REPORT_LEVEL` controls the amount of information presented in the infeasibility report. The default value is 1. For the above example the report is

```
MOSEK PRIMAL INFEASIBILITY REPORT.

Problem status: The problem is primal infeasible

The following constraints are involved in the primal infeasibility.

Index     Name      Lower bound      Upper bound      Dual lower       Dual upper
0         s0        NONE             2.000000e+002    0.000000e+000    1.000000e+000
2         s2        NONE             1.000000e+003    0.000000e+000    1.000000e+000
3         d1        1.100000e+003    1.100000e+003    1.000000e+000    0.000000e+000
4         d2        2.000000e+002    2.000000e+002    1.000000e+000    0.000000e+000

The following bound constraints are involved in the infeasibility.

Index     Name      Lower bound      Upper bound      Dual lower       Dual upper
8         x33       0.000000e+000    NONE             1.000000e+000    0.000000e+000
10        x34       0.000000e+000    NONE             1.000000e+000    0.000000e+000
```

The infeasibility report is divided into two sections corresponding to constraints and variables. It is a selection of those lines from the problem solution which are important in understanding primal infeasibility. In this case the constraints `s0`, `s2`, `d1`, `d2` and variables `x33`, `x34` are of importance because of nonzero dual values. The columns `Dual lower` and `Dual upper` contain the values of dual variables s_l^c, s_u^c, s_l^x and s_u^x in the primal infeasibility certificate (see Sec. 12.1.2).

In our example the certificate means that an appropriate linear combination of constraints `s0`, `s1` with coefficient $s_u^c = 1$, constraints `d1` and `d2` with coefficient $s_u^c - s_l^c = 0 - 1 = -1$ and lower bounds on `x33` and `x34` with coefficient $-s_l^x = -1$ gives a contradiction. Indeed, the combination of the four involved constraints is $x_{33} + x_{34} \leq -100$ (as indicated in the introduction, the difference between supply and demand).

It is also possible to extract the infeasible subproblem with the command-line tool. For an infeasible problem called `infeas.lp` the command:

```
mosek -d MSK_IPAR_INFEAS_REPORT_AUTO MSK_ON infeas.lp -info rinfeas.lp
```

will produce the file `rinfeas.bas.inf.lp` which contains the infeasible subproblem. Because of its size it may be easier to work with than the original problem file.

Returning to the transportation example, we discover that removing the fifth constraint $x_{12} = 200$ makes the problem feasible. Almost all undesired infeasibilities should be fixable at the modeling stage.

9.3.3 Locating dual infeasibility

A problem may also be *dual infeasible*. In this case the primal problem is usually unbounded, meaning that feasible solutions exists such that the objective tends towards infinity. For example, consider the problem

$$\begin{array}{ll} \text{maximize} & 200y_1 + 1000y_2 + 1000y_3 + 1100y_4 + 200y_5 + 500y_6 + 500y_7 \\ \text{subject to} & \qquad y_1 + y_4 \leq 1,\ y_1 + y_5 \leq 2,\ y_2 + y_6 \leq 5,\ y_2 + y_7 \leq 2, \\ & \qquad\qquad y_3 + y_4 \leq 1,\ y_3 + y_6 \leq 2,\ y_3 + y_7 \leq 1 \\ & \qquad\qquad\qquad y_1, y_2, y_3 \leq 0 \end{array}$$

which is dual to (9.1) (and therefore is dual infeasible). The dual infeasibility report may look as follows:

```
MOSEK DUAL INFEASIBILITY REPORT.

Problem status: The problem is dual infeasible

The following constraints are involved in the infeasibility.

Index     Name              Activity         Objective        Lower bound      Upper bound
5         x33               -1.000000e+00                     NONE             2.000000e+00
6         x34               -1.000000e+00                     NONE             1.000000e+00
```

(continues on next page)

(continued from previous page)

```
The following variables are involved in the infeasibility.

Index    Name              Activity         Objective        Lower bound      Upper bound
0        y1                -1.000000e+00    2.000000e+02     NONE             0.000000e+00
2        y3                -1.000000e+00    1.000000e+03     NONE             0.000000e+00
3        y4                1.000000e+00     1.100000e+03     NONE             NONE
4        y5                1.000000e+00     2.000000e+02     NONE             NONE

Interior-point solution summary
  Problem status  : DUAL_INFEASIBLE
  Solution status : DUAL_INFEASIBLE_CER
  Primal.  obj: 1.0000000000e+02    nrm: 1e+00    Viol.  con: 0e+00    var: 0e+00
```

In the report we see that the variables `y1`, `y3`, `y4`, `y5` and two constraints contribute to infeasibility with non-zero values in the `Activity` column. Therefore

$$(y_1, \ldots, y_7) = (-1, 0, -1, 1, 1, 0, 0)$$

is the dual infeasibility certificate as in Sec. 12.1.2. This just means, that along the ray

$$(0, 0, 0, 0, 0, 0, 0) + t(y_1, \ldots, y_7) = (-t, 0, -t, t, t, 0, 0), \ t > 0,$$

which belongs to the feasible set, the objective value $100t$ can be arbitrarily large, i.e. the problem is unbounded.

In the example problem we could

- Add a lower bound on `y3`. This will directly invalidate the certificate of dual infeasibility.
- Increase the objective coefficient of `y3`. Changing the coefficients sufficiently will invalidate the inequality $c^T y^* > 0$ and thus the certificate.

9.3.4 Suggestions

Primal infeasibility

When trying to understand what causes the unexpected primal infeasible status use the following hints:

- Remove the objective function. This does not change the infeasibility status but simplifies the problem, eliminating any possibility of issues related to the objective function.
- Remove cones, semidefinite variables and integer constraints. Solve only the linear part of the problem. Typical simple modeling errors will lead to infeasibility already at this stage.
- Consider whether your problem has some obvious necessary conditions for feasibility and examine if these are satisfied, e.g. total supply should be greater than or equal to total demand.
- Verify that coefficients and bounds are reasonably sized in your problem.
- See if there are any obvious contradictions, for instance a variable is bounded both in the variables and constraints section, and the bounds are contradictory.
- Consider replacing suspicious equality constraints by inequalities. For instance, instead of $x_{12} = 200$ see what happens for $x_{12} \geq 200$ or $x_{12} \leq 200$.
- Relax bounds of the suspicious constraints or variables.
- For integer problems, remove integrality constraints on some/all variables and see if the problem solves.

- Form an **elastic model**: allow to violate constraints at a cost. Introduce slack variables and add them to the objective as penalty. For instance, suppose we have a constraint

$$\begin{array}{ll} \text{minimize} & c^T x, \\ \text{subject to} & a^T x \leq b. \end{array}$$

  which might be causing infeasibility. Then create a new variable y and form the problem which contains:

$$\begin{array}{ll} \text{minimize} & c^T x + y, \\ \text{subject to} & a^T x \leq b + y. \end{array}$$

  Solving this problem will reveal by how much the constraint needs to be relaxed in order to become feasible. This is equivalent to inspecting the infeasibility certificate but may be more intuitive.

- If you think you have a feasible solution or its part, fix all or some of the variables to those values. Presolve will propagate them through the model and potentially reveal more localized sources of infeasibility.
- Dump the problem in OPF or LP format and verify that the problem that was passed to the optimizer corresponds to the problem expressed in the high-level modeling language, if any such was used.

Dual infeasibility

When trying to understand what causes the unexpected dual infeasible status use the following hints:

- Verify that the objective coefficients are reasonably sized.
- Check if no bounds and constraints are missing, for example if all variables that should be nonnegative have been declared as such etc.
- Strengthen bounds of the suspicious constraints or variables.
- Form an series of models with decreasing bounds on the objective, that is, instead of objective

$$\text{minimize } c^T x$$

  solve the problem with an additional constraint such as

$$c^T x = -10^5$$

  and inspect the solution to figure out the mechanism behind arbitrarily decreasing objective values. This is equivalent to inspecting the infeasibility certificate but may be more intuitive.

- Dump the problem in OPF or LP format and verify that the problem that was passed to the optimizer corresponds to the problem expressed in the high-level modeling language, if any such was used.

Please note that modifying the problem to invalidate the reported certificate does *not* imply that the problem becomes feasible — the reason for infeasibility may simply *move*, resulting a problem that is still infeasible, but for a different reason. More often, the reported certificate can be used to give a hint about errors or inconsistencies in the model that produced the problem.

9.4 Python Console

The **MOSEK** Python Console is an alternative to the **MOSEK** Command Line Tool. It can be used for interactive loading, solving and debugging optimization problems stored in files, for example **MOSEK** task files. It facilitates debugging techniques described in Sec. 9.

9.4.1 Usage

The tool requires Python 2 or 3. The **MOSEK** interface for Python must be installed following the installation instructions for Python API or Python Fusion API. In the basic case it should be sufficient to execute the script

```
python setup.py install --user
```

in the directory containing the **MOSEK** Python module.

The Python Console is contained in the file `mosekconsole.py` in the folder with **MOSEK** binaries. It can be copied to an arbitrary location. The file is also available for `download here (mosekconsole.py)`.

To run the console in interactive mode use

```
python mosekconsole.py
```

To run the console in batch mode provide a semicolon-separated list of commands as the second argument of the script, for example:

```
python mosekconsole.py "read data.task.gz; solve form=dual; writesol data"
```

The script is written using the **MOSEK** Python API and can be extended by the user if more specific functionality is required. We refer to the documentation of the Python API.

9.4.2 Examples

To read a problem from `data.task.gz`, solve it, and write solutions to `data.sol`, `data.bas` or `data.itg`:

```
read data.task.gz; solve; writesol data
```

To convert between file formats:

```
read data.task.gz; write data.mps
```

To set a parameter before solving:

```
read data.task.gz; param INTPNT_CO_TOL_DFEAS 1e-9; solve"
```

To list parameter values related to the mixed-integer optimizer in the task file:

```
read data.task.gz; param MIO
```

To print a summary of problem structure:

```
read data.task.gz; anapro
```

To solve a problem forcing the dual and switching off presolve:

```
read data.task.gz; solve form=dual presolve=no
```

To write an infeasible subproblem to a file for debugging purposes:

```
read data.task.gz; solve; infsub; write inf.opf
```

9.4.3 Full list of commands

Below is a brief description of all the available commands. Detailed information about a specific command `cmd` and its options can be obtained with

```
help cmd
```

Table 9.1: List of commands of the MOSEK Python Console.

Command	Description
`help [command]`	Print list of commands or info about a specific command
`log filename`	Save the session to a file
`intro`	Print MOSEK splashscreen
`testlic`	Test the license system
`read filename`	Load problem from file
`reread`	Reload last problem file
`solve [options]`	Solve current problem
`write filename`	Write current problem to file
`param [name [value]]`	Set a parameter or get parameter values
`paramdef`	Set all parameters to default values
`info [name]`	Get an information item
`anapro`	Analyze problem data
`hist`	Plot a histogram of problem data
`histsol`	Plot a histogram of the solutions
`spy`	Plot the sparsity pattern of the A matrix
`truncate epsilon`	Truncate small coefficients down to 0
`anasol`	Analyze solutions
`removeitg`	Remove integrality constraints
`infsub`	Replace current problem with its infeasible subproblem
`writesol basename`	Write solution(s) to file(s) with given basename
`delsol`	Remove all solutions from the task
`exit`	Leave

Chapter 10

Technical guidelines

This section contains some more in-depth technical guidelines for Fusion API for Python, not strictly necessary for basic use of **MOSEK**.

10.1 Limitations

Fusion imposes some limitations on certain aspects of a model to ensure easier portability:

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

The limits on the model size in *Fusion* are as follows:

- The maximum number of variable elements is $2^{31} - 1$.
- The maximum size of a dimension is $2^{31} - 1$.
- The total size of an item (the product of dimensions) is limited to $2^{63} - 1$.

10.2 Memory management and garbage collection

Users who experience memory leaks using *Fusion*, especially:

- memory usage not decreasing after the solver terminates,
- memory usage increasing when solving a sequence of problems,

should make sure that the `Model` objects are properly garbage collected. Since each `Model` object links to a **MOSEK** task resource in a linked library, it is sometimes the case that the garbage collector is unable to reclaim it automatically. This means that substantial amounts of memory may be leaked. For this reason it is very important to make sure that the `Model` object is disposed of manually when it is not used any more. The necessary cleanup is performed by the method `Model.dispose`.

The `Model` supports the *Context Manager* protocol, so it will be destroyed properly when used in the construction:

```
with Model() as M:
    # Work with the model here
    pass;
```

One can also write

```
try:
    M = Model()
    # Work with the model here
finally:
    M.dispose()
```

This construction assures that the *Model.dispose* method is called when the object goes out of scope, even if an exception occurred. If this approach cannot be used, e.g. if the *Model* object is returned by a factory function, one should explicitly call the *Model.dispose* method when the object is no longer used.

Furthermore, if the *Model* class is extended, it is necessary to dispose of the superclass if the initialization of the derived subclass fails. One can use a construction such as:

```
class MyModel(Model):
    def __init__(self):
        finished = False
        try:
            Model.__init__(self)
            # other initialization
            finished = True
        finally:
            if not finished:
                self.dispose()
```

10.3 Names

All elements of an optimization problem in **MOSEK** (objective, constraints, variables, etc.) can be given names. Assigning meaningful names to variables and constraints makes it much easier to understand and debug optimization problems dumped to a file. On the other hand, note that assigning names can substantially increase setup time, so it should be avoided in time-critical applications.

The *Model* object's, variables' and constraints' constructors provide versions with a string name as an optional parameter.

10.4 Multithreading

Thread safety

Sharing a *Model* object between threads is safe, as long as it is not accessed from more than one thread at a time. Multiple *Model* objects can be used in parallel without any problems.

Parallelization

The interior-point and mixed-integer optimizers in **MOSEK** are parallelized. By default **MOSEK** will automatically select the number of threads. However, the maximum number of threads allowed can be changed by setting the parameter *numThreads* and related parameters. This should never exceed the number of cores. See Sec. 13 and Sec. 13.4 for more details.

The speed-up obtained when using multiple threads is highly problem and hardware dependent. We recommend experimenting with various thread numbers to determine the optimal settings. For small problems using multiple threads may be counter-productive because of the associated overhead.

Determinism

By default the optimizer is run-to-run deterministic, which means that it will return the same answer each time it is run on the same machine with the same input, the same parameter settings (including number of threads) and no time limits.

Setting the number of threads

The number of threads the optimizer uses can be changed with the parameter *numThreads*.

For conic problems (when the conic optimizer is used) the value set at the first optimization will remain fixed through the lifetime of the process. The thread pool will be reserved once for all and subsequent changes to *numThreads* will have no effect. The only possibility at that point is to switch between multi-threaded and single-threaded interior-point optimization using the parameter *intpntMultiThread*.

The parameter *numThreads* affects only the optimizer. It may be the case that `numpy` is consuming more threads. In most cases this can be limited by setting the environment variable `MKL_NUM_THREADS`. See the `numpy` documentation for more details.

10.5 Efficiency

In some cases *Fusion* must reformulate the problem by adding auxiliary variables and constraints before it can be represented in the optimizer's internal format. This can cause a significant overhead. The following guidelines can help speed up the process.

Decide between sparse and dense matrices

Deciding whether a matrix should be stored in dense or sparse format is not always trivial. First, there are storage considerations. An $n \times m$ matrix with l non zero entries, requires

- $\approx n \cdot m$ storage space in dense format,
- $\approx 3 \cdot l$ storage space in sparse (triplet) format.

Therefore if $l \ll n \cdot m$, then the sparse format has smaller memory requirements. Especially for very sparse density matrices it will also yield much faster expression transformations. Also, this is the format used ultimately by the underlying optimizer task. However, there are borderline cases in which these advantages may vanish due to overhead spent creating the triplet representation.

Sparsity is a key feature of many optimization models and often occurs 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 of choosing the most appropriate storage format.

Reduce the number of *Fusion* calls and level of nesting

A possible source of performance degradation is an excessive use of nested expressions resulting in a large number of *Fusion* calls with small model updates, where instead the model could be updated in larger chunks at once. In general, `loop`-free code and reduction of expression nesting are likely to be more efficient. For example the expression

$$\sum_{i=1}^{n} A_i x_i$$
$$x_i \in \mathbb{R}^k, A_i \in \mathbb{R}^{k \times k},$$

could be implemented 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 the intermediate expressions for $A_i x_i$ and sum all of them in one step:

```
ee = Expr.add( [ Expr.mul(AA,xx) for (AA,xx) in zip(A,x)] )
```

Fusion design naturally promotes this sort of vectorized implementations. See Sec. 6.7 for more examples.

Do not fetch the whole solution if not necessary

Fetching a solution from a shaped variable produces a flat array of values. This means that some reshaping has to take place and that the user gets all values even if they are potentially interested only in some of them. In this case, it is better to create a slice variable holding the relevant elements and fetch the solution for this subset. See Sec. 6.6. Fetching the full solution may cause an exception due to memory exhaustion or platform-dependent constraints on array sizes.

Remove names

Variables, constraints and the objective function can be constructed with user-assigned names. While this feature is very useful for debugging and improves the readability of both the code and of problems dumped to files, it also introduces quite some overhead: *Fusion* must check and make sure that names are unique. For optimal performance it is therefore recommended to not specify names at all.

10.6 The license system

MOSEK is a commercial product that **always** needs a valid license to work. **MOSEK** uses a third party license manager to implement license checking. The number of license tokens provided determines the number of optimizations that can be run simultaneously.

By default a license token remains checked out from the first optimization until the end of the **MOSEK** session, i.e.

- a license token is checked out when the method *Model.solve* is called the first time, and
- the token is returned when the process exits.

Starting the optimization when no license tokens are available will result in an error.

Default behaviour of the license system can be changed in several ways:

- Setting the parameter *cacheLicense* to *"off"* will force **MOSEK** to return the license token immediately after the optimization completed.
- Setting the license wait flag with *Model.putlicensewait* or with the parameter *licenseWait* will force **MOSEK** to wait until a license token becomes available instead of throwing an exception.
- Additional license checkouts and checkins can be performed manually through the underlying **MOSEK** task and environment. See Sec. 8.8.
- The default path to the license file can be changed with *Model.putlicensepath*.

10.7 Deployment

When redistributing a Python application using the **MOSEK** Fusion API for Python 9.0.98, the following libraries must be included:

64-bit Linux	64-bit Windows	32-bit Windows	64-bit Mac OS
`libmosek64.so.9.0`	`mosek64_9_0.dll`	`mosek9_0.dll`	`libmosek64.9.0.dylib`
`libcilkrts.so.5`	`cilkrts20.dll`	`cilkrts20.dll`	`libcilkrts.5.dylib`
`libmosekxx9_0.so`	`mosekxx9_0.dll`	`mosekxx9_0.dll`	`libmosekxx9_0.dylib`

Furthermore, one (or both) of the directories

- `python/2/mosek` for Python 2.x applications,
- `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`. Alternatively, if the binary libraries reside in another directory, the application can pre-load the `mosekxx` library from another location before `mosek` is imported, e.g. like this

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

Chapter 11

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 of Sec. 7 before going through these advanced case studies.

- *Portfolio Optimization*
  - **Keywords:** Markowitz model, variance, risk, efficient frontier, transaction cost, market impact cost, cardinality constraints
  - **Type:** Conic Quadratic, Power Cone, Mixed-Integer
- *Primal SVM*
  - **Keywords:** machine learning, Support-Vector Machine, hyperplane separation, classifier
  - **Type:** Conic Quadratic
- *2D Total Variation*
  - **Keywords:** denoising, total variation
  - **Type:** Conic Quadratic
- *Multi Processor Scheduling*
  - **Keywords:** scheduling, job allocation, feasible point heuristic
  - **Type:** Mixed-Integer, Linear Optimization
- *Logistic regression*
  - **Keywords:** machine learning, logistic regression, classifier, log-sum-exp, softplus, regularization
  - **Type:** Exponential Cone, Quadratic Cone
- *Inner and outer Löwner-John Ellipsoids*
  - **Keywords:** volume optimization, ellipsoidal approximation, determinant, geometric mean, eigenvalues
  - **Type:** Power Cone, Semidefinite
- *SUDOKU*
  - **Keywords:** combinatorial puzzle, binary variables, integer modeling
  - **Type:** Integer Optimization, Linear Optimization
- *Travelling Salesman*
  - **Keywords:** TSP, cycle elimination
  - **Type:** Mixed-Integer, Linear Optimization

- *Nearest Correlation Matrix Problem*
  - **Keywords:** low-rank matrix approximation, trace, Frobenius norm, correlation matrix
  - **Type:** Semidefinite
- *Semidefinite relaxation of MIQCQO problems*
  - **Keywords:** integer quadratic problems, semidefinite relaxation, approximation, integer least squares
  - **Type:** Semidefinite, Mixed-Integer Conic Quadratic

11.1 Portfolio Optimization

In this section the Markowitz portfolio optimization problem and variants are implemented using Fusion API for Python.

- *Basic Markowitz model*
- *Efficient frontier*
- *Factor model and efficiency*
- *Market impact costs*
- *Transaction costs*
- *Cardinality constraints*

11.1.1 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

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

The standard deviation

$$\sqrt{x^T \Sigma x}$$

is usually associated with risk.

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

$$\begin{array}{lrcl} \mbox{maximize} & \mu^T x & & \\ \mbox{subject to} & e^T x & = & w + e^T x^0, \\ & x^T \Sigma x & \leq & \gamma^2, \\ & x & \geq & 0. \end{array} \tag{11.1}$$

The variables x denote 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 interpreted 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, is bounded by the parameter γ^2. Therefore, γ specifies an upper bound of the standard deviation (risk) 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

$$\Sigma = GG^T. \tag{11.2}$$

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 Sec. 11.1.3. 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

$$(\gamma, G^T x) \in \mathcal{Q}^{n+1},$$

where $\mathcal{Q}^{n+1}$ is the $(n+1)$-dimensional quadratic cone. Therefore, problem (11.1) can be written as

$$\begin{array}{lrcl} \text{maximize} & \mu^T x & & \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & (\gamma, G^T x) & \in & \mathcal{Q}^{n+1}, \\ & x & \geq & 0, \end{array} \tag{11.3}$$

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

$$\mu = \begin{bmatrix} 0.1073 \\ 0.0737 \\ 0.0627 \end{bmatrix}$$

and

$$\Sigma = 0.1 \cdot \begin{bmatrix} 0.2778 & 0.0387 & 0.0021 \\ 0.0387 & 0.1112 & -0.0020 \\ 0.0021 & -0.0020 & 0.0115 \end{bmatrix}.$$

This implies

$$G^T = \sqrt{0.1} \begin{bmatrix} 0.5271 & 0.0734 & 0.0040 \\ 0 & 0.3253 & -0.0070 \\ 0 & 0 & 0.1069 \end{bmatrix}$$

Why a Conic Formulation?

Problem (11.1) is a convex quadratically constrained optimization problem that can be solved directly using **MOSEK**. Why then reformulate it as a conic quadratic optimization problem (11.3)? The main reason for choosing a conic model is that it is more robust and usually solves faster and more reliably. For instance it is not always easy to numerically validate that the matrix Σ in (11.1) is positive semidefinite due to the presence of rounding errors. It is also very easy to make a mistake so Σ becomes indefinite. These problems are completely eliminated in the conic formulation.

Moreover, observe the constraint

$$\|G^T x\| \leq \gamma$$

more numerically robust than

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

for very small and very large values of γ. Indeed, if say $\gamma \approx 10^4$ then $\gamma^2 \approx 10^8$, which introduces a scaling issue in the model. Hence, using conic formulation we work with the standard deviation instead of variance, which usually gives rise to a better scaled model.

Example code

Listing 11.1 demonstrates how the basic Markowitz model (11.3) is implemented.

Listing 11.1: Code implementing problem (11.3).

```
def BasicMarkowitz(n,mu,GT,x0,w,gamma):

    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)))
```

(continues on next page)

(continued from previous page)

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

        # Solves the model.
        M.solve()

        return np.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

$$\left(\gamma, G^T x\right)$$

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

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

11.1.2 The Efficient Frontier

The portfolio computed by the Markowitz model is efficient in the sense that there is no 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 with all efficient portfolios so the investor can choose the portfolio that has the desired tradeoff between return and risk.

Given a nonnegative α the problem

$$\begin{array}{ll} \text{maximize} & \mu^T x - \alpha x^T \Sigma x \\ \text{subject to} & e^T x = w + e^T x^0, \\ & x \geq 0. \end{array} \tag{11.4}$$

is one standard way to trade the expected return against penalizing variance. Note that, in contrast to the previous example, we explicitly use the variance ($\|G^T x\|_2^2$) rather than standard deviation ($\|G^T x\|_2$), therefore the conic model includes a rotated quadratic cone:

$$\begin{array}{lrcll} \text{maximize} & \mu^T x - \alpha s & & & \\ \text{subject to} & e^T x & = & w + e^T x^0, & \\ & (s, 0.5, G^T x) & \in & Q_r^{n+2} & (\text{equiv. to } s \geq \|G^T x\|_2^2 = x^T \Sigma x), \\ & x & \geq & 0. & \end{array} \tag{11.5}$$

The parameter α specifies the tradeoff between expected return and variance. Ideally the problem (11.4) should be solved for all values $\alpha \geq 0$ but in practice it is impossible. Using the example data from Sec. 11.1.1, the optimal values of return and variance for several values of α are shown in the figure.

Example code

Listing 11.2 demonstrates how to compute the efficient portfolios for several values of α.

Listing 11.2: Code for the computation of the efficient frontier based on problem (11.4).

```
def EfficientFrontier(n,mu,GT,x0,w,alphas):

    with Model("Efficient frontier") as M:

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

(continues on next page)

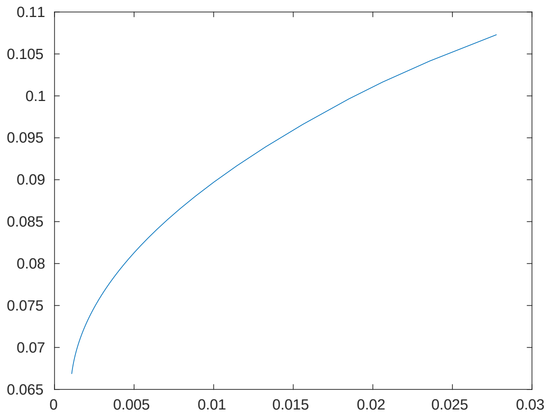


Fig. 11.1: The efficient frontier for the sample data.

(continued from previous page)

```
    # 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())      # Variance variable

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

    # Computes the risk
    M.constraint('variance', Expr.vstack(s, 0.5, Expr.mul(GT,x)), Domain.inRotatedQCone())

    frontier = []

    mudotx = Expr.dot(mu,x)

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

        M.solve()

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

    return frontier
```

Note the efficient frontier could also have been computed using the code in Sec. 11.1.1 by varying γ. However, when the constraints of a *Fusion* model are changed the model has to be rebuilt whereas a rebuild is not needed if only the objective is modified.

11.1.3 Factor model and efficiency

In practice it is often important to solve the portfolio problem very quickly. Therefore, in this section we discuss how to improve computational efficiency at the modeling stage.

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 sparsity: the number of nonzeros used to represent the problem. Indeed it is often better to focus on the number of nonzeros in G see (11.2) 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 and usually p is much smaller than n. In practice p tends to 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

$$G^T = \begin{bmatrix} D^{1/2} \\ V^T \end{bmatrix}$$

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 storage requirements are reduced by 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 matrix 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. More details about this process can be found in *[And13]*.

11.1.4 Slippage Cost

The basic Markowitz model assumes that there are no costs associated with trading the assets and that the returns of the assets are independent of the amount traded. Neither of those assumptions is usually valid in practice. Therefore, a more realistic model is

$$
\begin{array}{lrcl}
\text{maximize} & \mu^T x & & \\
\text{subject to} & e^T x + \sum_{j=1}^n T_j(\Delta x_j) & = & w + e^T x^0, \\
& x^T \Sigma x & \leq & \gamma^2, \\
& x & \geq & 0.
\end{array} \tag{11.6}
$$

Here Δx_j is the change in the holding of asset j i.e.

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

and $T_j(\Delta x_j)$ specifies the transaction costs when the holding of asset j is changed from its initial value. In the next two sections we show two different variants of this problem with two nonlinear cost functions T.

11.1.5 Market Impact Costs

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

$$T_j(\Delta x_j) = m_j |\Delta x_j|^{3/2}$$

where m_j is a constant that is estimated in some way by the trader. See *[GK00]* [p. 452] for details. From the Modeling Cookbook we know that $t \geq |z|^{3/2}$ can be modeled directly using the power cone $\mathcal{P}_3^{2/3,1/3}$:

$$\{(t,z) : t \geq |z|^{3/2}\} = \{(t,z) : (t,1,z) \in \mathcal{P}_3^{2/3,1/3}\}$$

Hence, it follows that $\sum_{j=1}^n T_j(\Delta x_j) = \sum_{j=1}^n m_j |x_j - x_j^0|^{3/2}$ can be modeled by $\sum_{j=1}^n m_j t_j$ under the constraints

$$
\begin{array}{rl}
z_j & = |x_j - x_j^0|, \\
(t_j, 1, z_j) & \in \mathcal{P}_3^{2/3,1/3}.
\end{array}
$$

Unfortunately this set of constraints is nonconvex due to the constraint

$$z_j = |x_j - x_j^0| \tag{11.7}$$

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

$$z_j \geq |x_j - x_j^0|, \tag{11.8}$$

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

$$z_j > |x_j - x_j^0| \tag{11.9}$$

cannot hold for an optimal solution.

If the optimal solution has the property (11.9) then the market impact cost 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 the only way to obtain a small risk is to get rid of some of the assets by generating transaction costs. We generally assume that this is not the case and hence the models (11.7) and (11.8) are equivalent.

The above observations lead to

$$
\begin{array}{lrcll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + m^T t & = & w + e^T x^0, & \\
 & (\gamma, G^T x) & \in & \mathcal{Q}^{n+1}, & \\
 & (t_j, 1, x_j - x_j^0) & \in & \mathcal{P}_3^{2/3,1/3}, & j = 1, \ldots, n, \\
 & x & \geq & 0. &
\end{array} \tag{11.10}
$$

The revised budget constraint

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

specifies that the initial wealth covers the investment and the transaction costs. It should be mentioned that transaction costs of the form

$$t_j \geq |z_j|^p$$

where $p > 1$ is a real number can be modeled with the power cone as

$$(t_j, 1, z_j) \in \mathcal{P}_3^{1/p, 1-1/p}.$$

See the Modeling Cookbook for details.

Example code

Listing 11.3 demonstrates how to compute an optimal portfolio when market impact cost are included.

Listing 11.3: Implementation of model (11.10).

```
def MarkowitzWithMarketImpact(n,mu,GT,x0,w,gamma,m):
    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))

        # Variables computing market impact
        t = M.variable("t", 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())

        # t >= |x-x0|^1.5 using a power cone
        M.constraint('tz', Expr.hstack(t, Expr.constTerm(n, 1.0), Expr.sub(x,x0)), Domain.
↪inPPowerCone(2.0/3.0))

        M.solve()

        return x.level(), t.level()
```

11.1.6 Transaction Costs

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

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

Hence, whenever asset j is traded we pay a fixed setup cost f_j and a variable cost of g_j per unit traded. Given the assumptions about transaction costs in this section problem (11.6) may be formulated as

$$
\begin{array}{lrcll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + f^T y + g^T z & = & w + e^T x^0, & \\
& (\gamma, G^T x) & \in & \mathcal{Q}^{n+1}, & \\
& z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
& z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
& z_j & \leq & U_j y_j, & j = 1, \ldots, n, \\
& y_j & \in & \{0, 1\}, & j = 1, \ldots, n, \\
& x & \geq & 0. &
\end{array}
\tag{11.11}
$$

First observe that

$$z_j \geq |x_j - x_j^0| = |\Delta x_j|.$$

We choose U_j as some a priori 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 cost for asset j is given by

$$f_j y_j + g_j z_j.$$

Example code

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

Listing 11.4: Code solving problem (11.11).

```
def MarkowitzWithTransactionsCost(n,mu,GT,x0,w,gamma,f,g):
    # 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())
```

(continues on next page)

(continued from previous page)

```
        # 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()

        return x.level(), y.level(), z.level()
```

11.1.7 Cardinality constraints

Another method to reduce costs involved with processing transactions is to only change positions in a small number of assets. In other words, at most k of the differences $|\Delta x_j| = |x_j - x_j^0|$ are allowed to be non-zero, where k is (much) smaller than the total number of assets n.

This type of constraint can be again modeled by introducing a binary variable y_j which indicates if $\Delta x_j \neq 0$ and bounding the sum of y_j. The basic Markowitz model then gets updated as follows:

$$
\begin{array}{lrcll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x & = & w + e^T x^0, & \\
 & (\gamma, G^T x) & \in & \mathcal{Q}^{n+1}, & \\
 & z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
 & z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
 & z_j & \leq & U_j y_j, & j = 1, \ldots, n, \\
 & y_j & \in & \{0, 1\}, & j = 1, \ldots, n, \\
 & e^T y & \leq & k, & \\
 & x & \geq & 0, &
\end{array}
\tag{11.12}
$$

were U_j is some a priori chosen upper bound on the amount of trading in asset j.

Example code

The following example code demonstrates how to compute an optimal portfolio with cardinality bounds.

Listing 11.5: Code solving problem (11.12).

```
def MarkowitzWithCardinality(n,mu,GT,x0,w,gamma,k):
    # Upper bound on the traded amount
    w0 = w+sum(x0)
    u = n*[w0]

    with Model("Markowitz portfolio with cardinality bound") 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  - do we change position in assets
        y = M.variable("y", n, Domain.binary())

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

(continues on next page)

(continued from previous page)

```
        # 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())

        # 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))

        # 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))

        # At most k assets change position
        M.constraint('cardinality', Expr.sum(y), Domain.lessThan(k))

        # 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()

        return x.level()
```

If we solve our running example with $k = 1, 2, 3$ then we get the following solutions, with increasing expected returns:

```
Bound:  1  Expected return: 0,0627   Solution: 0,0000 0,0000 1,0000
Bound:  2  Expected return: 0,0669   Solution: 0,0939 0,0000 0,9061
Bound:  3  Expected return: 0,0685   Solution: 0,1010 0,1156 0,7834
```

11.2 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 study focuses on the Support-Vector Machines (SVM).

The basic SVM model can be stated as:

> We are given a set of m points in $\mathbb{R}^n$, partitioned into 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

Let $x_1, \ldots, x_m \in \mathbb{R}^n$ be the given training set and let $y_i \in \{-1, +1\}$ be the labels indicating the group membership of the i-th training example. Then we want to determine an affine hyperplane $w^T x = b$ that separates the group in the strong sense that

$$y_i(w^T x_i - b) \geq 1 \tag{11.13}$$

for all i, the property referred to as *large margin classification*: the strip $\{x \in \mathbb{R}^n \;:\; -1 < w^T x - b < 1\}$ does not contain any training example. The width of this strip is $2\|w\|^{-1}$, and maximizing that quantity is equivalent to minimizing $\|w\|$. We get that the large margin classification is the solution of the following optimization problem:

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

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 admits a violation of the large margin requirement (11.13) by a non-negative slack variable which is then penalized in the objective function.

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

In matrix form we have

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

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-offs 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 involves converting the quadratic term of the objective function into a conic constraint:

$$
\begin{array}{ll}
\text{minimize}_{b,w,\xi,t} & t + C\mathbf{e}^T\xi \\
\text{subject to} & \xi + y \star (Xw - b\mathbf{e}) \geq \mathbf{e}, \\
 & (1, t, w) \in \mathcal{Q}_r^{n+2}, \\
 & \xi \geq 0.
\end{array}
\tag{11.14}
$$

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

Fusion implementation

We now demonstrate how implement model (11.14). Let us assume that the training examples are stored in the rows of a matrix `X`, the labels in a vector `y` and that we have a set of weights `C` for which we want to train the model. The implementation in *Fusion* of our conic model starts declaring the model class:

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

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 by 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:

1. element-wise multiplication $\star$ is performed with the *Expr.mulElm* function;
2. a vector whose entries are repetitions of b is produced by *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) ) ) )
```

To solve a sequence of problems with varying `C` we can simply iterate 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()
```

Source code

Listing 11.6: The code implementing model (11.14)

```
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() )

        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;
```

Example

We generate a random dataset consisting of two groups of points, each from a Gaussian distribution in $\mathbb{R}^2$ with centres $(1.0, 1.0)$ and $(-1.0, -1.0)$, respectively.

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

    m  = 50
    n  = 3
```

(continues on next page)

(continued from previous page)

```
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)]
```

With standard deviation $\sigma = 1/2$ we obtain a separable instance of the problem with a solution shown in Fig. 11.2.

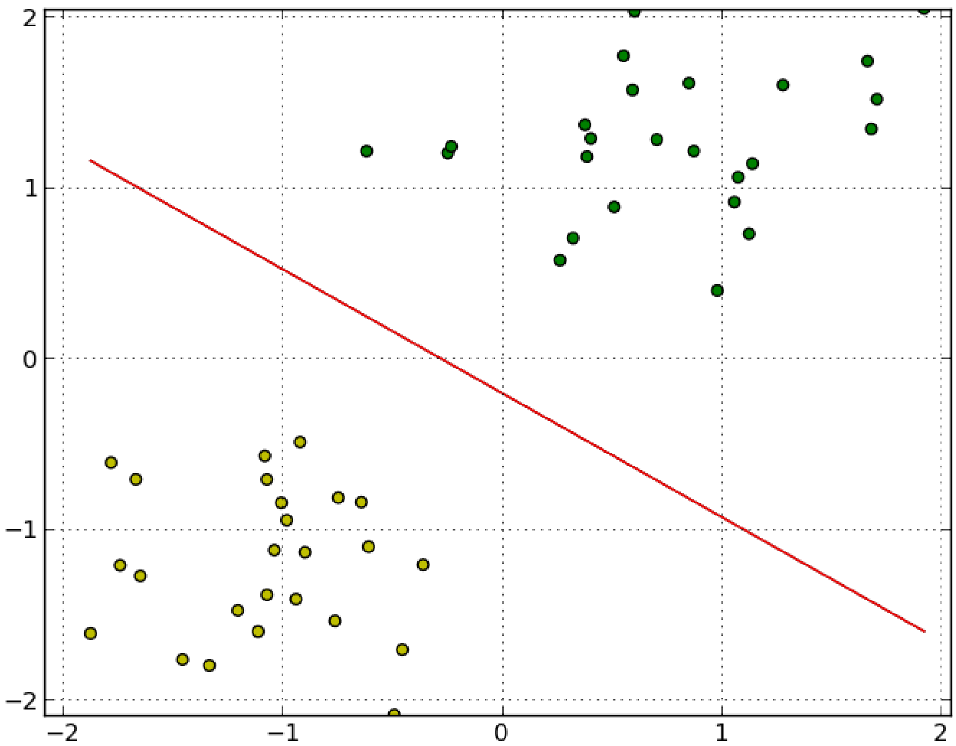


Fig. 11.2: Separating hyperplane for two clusters of points.

For $\sigma = 1$ the two groups are not linearly separable and the we obtain the optimal hyperplane as in Fig. 11.3.

11.3 2D Total Variation

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

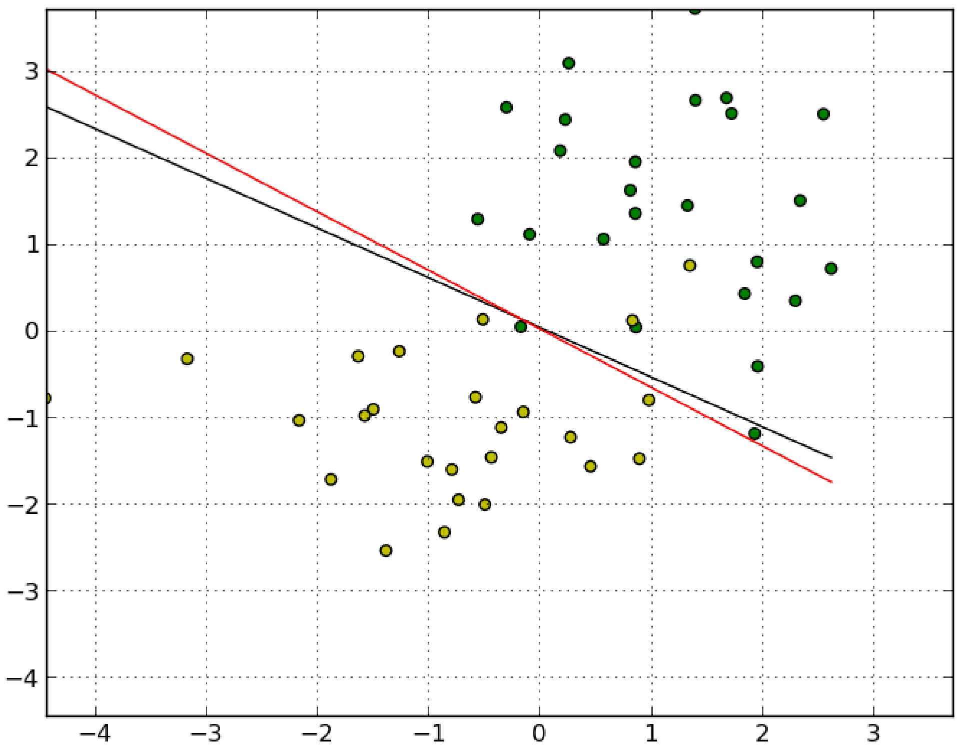


Fig. 11.3: Soft separating hyperplane for two groups of points.

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 $u_{ij} \in [0, 1]$ is the actual signal value and v_{ij} is the noise. The aim is to reconstruct u 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 translates into a simple conic quadratic constraint as

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

We aim to minimize the change in signal value when moving between adjacent cells. To this end we define the adjacent differences vector as

$$\partial_{ij}^+ = \begin{pmatrix} \partial_{ij}^x \\ \partial_{ij}^y \end{pmatrix} = \begin{pmatrix} u_{i+1,j} - u_{i,j} \\ u_{i,j+1} - u_{i,j} \end{pmatrix}, \tag{11.15}$$

for each cell $1 \leq i, j \leq n$ (we assume that the respective coordinates ∂_{ij}^x and ∂_{ij}^y are zero on the right and bottom boundary of the grid).

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

$$t_{ij} \geq \|\partial_{ij}^+\|_2 \quad \text{or} \quad \left(t_{ij}, \partial_{ij}^+\right) \in \mathcal{Q},$$

and minimize the sum of all t_{ij}.

The complete model takes the form:

$$\begin{array}{lll} \min & \sum_{1 \leq i,j \leq n} t_{ij}, & \\ \text{s.t.} & \partial_{ij}^+ = \left(u_{i+1,j} - u_{i,j}, u_{i,j+1} - u_{i,j}\right)^T, & \forall 1 \leq i, j \leq n, \\ & \left(t_{ij}, \partial_{ij}^+\right) \in \mathcal{Q}^3, & \forall 1 \leq i, j \leq n, \\ & (\sigma, u - f) \in \mathcal{Q}^{nm+1}, & \\ & u_{i,j} \in [0, 1]. & \forall 1 \leq i, j \leq n. \end{array} \tag{11.16}$$

Implementation

The *Fusion* implementation of model (11.16) uses variable and expression slices.

First of all we start by 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 the dimensions of `u` is larger than those of the grid to accommodate the boundary conditions later. The actual cells of the grid are defined as a slice of `u`:

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

The next step is to define the partial variation along each axis, as in (11.15):

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

Slices are created on the fly as they will not be reused. Now we can set the conic constraints on the norm of the total variations. To this extent we stack the variables `t`, `deltax` and `deltay` together and demand that each row of the new matrix is in a quadratic cone.

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

We now need to bound the norm of the noise. This can be achieved with a conic constraint using `f` as a one-dimensional array:

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

The objective function is the sum of all t_{ij}:

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

Example

Consider the linear signal $u_{ij} = \frac{i+j}{n+m}$ and its modification with random Gaussian noise, as in Fig. 11.4. Various reconstructions of u, obtained with different values of σ, are shown in Fig. 11.5 (where $\bar{\sigma} = \sigma/nm$ is the relative noise bound per cell).

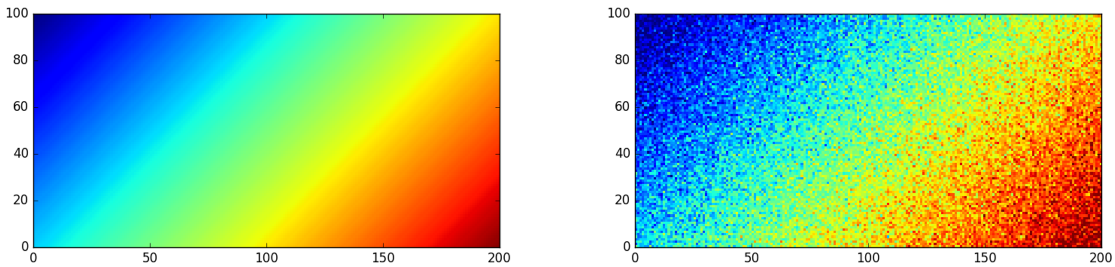


Fig. 11.4: A linear signal and its modification with random Gaussian noise.

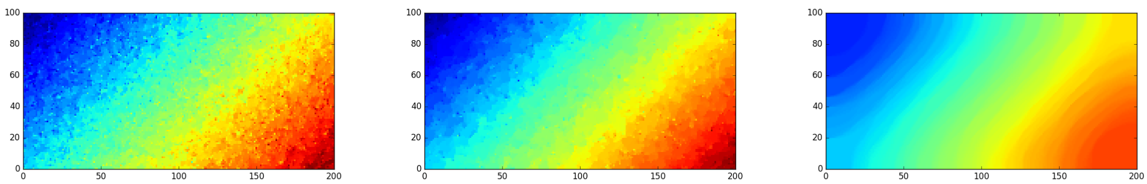


Fig. 11.5: Three reconstructions of the linear signal obtained for $\bar{\sigma} \in \{0.0004, 0.0005, 0.0006\}$, respectively.

Source code

Listing 11.7: The *Fusion* implementation of model (11.16).

```
def total_var(n,m,f,sigma):
    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() )

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

        return ucore.level()
```

11.4 Multiprocessor Scheduling

In this case 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

- to show how to define a Integer Linear Programming model,
- to take advantage of *Fusion* operators to compactly express sets of constraints,
- to provide the solver with 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 completion time among all machines.

Formally, we introduce a binary variable x_{ij} that takes value 1 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:

$$\begin{array}{lll} & \min \max_{i\in M} \sum_{j\in J} T_j x_{ij} & \\ \text{s.t.} & \sum_{i\in M} x_{ij} = 1, & j \in J, \\ & x_{ij} \in \{0,1\} & \forall i \in M, j \in J. \end{array} \tag{11.17}$$

Model (11.17) can be easily transformed into an integer linear programming model as follows:

$$\begin{array}{lll} & \min t & \\ \text{s.t.} & \sum_{i\in M} x_{ij} = 1, & j \in J, \\ & t \geq \sum_{j\in J} T_j x_{ij}, & i \in M, \\ & x_{ij} \in \{0,1\}, & \forall i \in M, j \in J. \end{array} \tag{11.18}$$

The implementation of this model in *Fusion* is straightforward:

```python
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-explanatory. The only critical point is

```python
    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

$$t\mathbf{1} - xT \geq 0. \tag{11.19}$$

The function *Var.repeat* creates a vector of length m, as required for (11.19). The same result can be obtained via matrix multiplication, i.e. using *Expr.mul*, but in this particular case *Var.repeat* is faster as it only performs a logical operation.

Longest Processing Time first rule (LPT)

The multiprocessor scheduling is known to be an NP-complete problem (see *[GJ79]*). Nevertheless there are effective heuristics, with provable worst case bounds, that are able to provide 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` be 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 the load of machine `k`,
  - remove `i` from `M`.

This simple algorithm is a $\frac{1}{3}(4 - \frac{1}{m})$ approximation. So for $m = 1$ we get the optimal solution (indeed there is no choice with a single machine); for $m \to \infty$ the approximation factor is no worse than 4/3 (again see *[Gra69]*).

A simple implementation is given below.

```python
    #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 part 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)
```

We can test the program with and without providing the initial LPT solution. Our random datasets consists of a mix of tasks with long and short processing times and we accept a solution at relative optimality tolerance 0.01. Some results are shown in the table below.

Table 11.1: Sample test results for the makespan problem.

n	m	long tasks	short tasks	No LPT	With LPT
1000	8	20%	80%	13.36s	1.23s
1000	8	80%	20%	1.35s	1.24s
100	12	20%	80%	16.37s	0.11s
100	12	80%	20%	16.62s	10.01s
20	20	0%	100%	10.38s	21.88s

We can see that depending on the structure and parameters of the problem it may pay off to provide an initial LPT solution. Therefore it is always recommended to test the mixed-integer solver with different settings to find the most efficient setup for a given problem.

Listing 11.8: Complete code for the LPT scheduling example.

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


def main():
    #Parameters:
    n = 30                  #Number of tasks
    m = 6                   #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
```

(continues on next page)

(continued from previous page)

```
        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()
```

11.5 Logistic regression

Logistic regression is an example of a binary classifier, where the output takes one two values 0 or 1 for each data point. We call the two values *classes*.

Formulation as an optimization problem

Define the sigmoid function

$$S(x) = \frac{1}{1+\exp(-x)}.$$

Next, given an observation $x \in \mathbb{R}^d$ and a weights $\theta \in \mathbb{R}^d$ we set

$$h_\theta(x) = S(\theta^T x) = \frac{1}{1+\exp(-\theta^T x)}.$$

The weights vector θ is part of the setup of the classifier. The expression $h_\theta(x)$ is interpreted as the probability that x belongs to class 1. When asked to classify x the returned answer is

$$x \mapsto \begin{cases} 1 & h_\theta(x) \geq 1/2, \\ 0 & h_\theta(x) < 1/2. \end{cases}$$

When training a logistic regression algorithm we are given a sequence of training examples x_i, each labelled with its class $y_i \in \{0,1\}$ and we seek to find the weights θ which maximize the likelihood function

$$\prod_i h_\theta(x_i)^{y_i}(1-h_\theta(x_i))^{1-y_i}.$$

Of course every single y_i equals 0 or 1, so just one factor appears in the product for each training data point. By taking logarithms we can define the logistic loss function:

$$J(\theta) = -\sum_{i:y_i=1} \log(h_\theta(x_i)) - \sum_{i:y_i=0} \log(1-h_\theta(x_i)).$$

The training problem with regularization (a standard technique to prevent overfitting) is now equivalent to

$$\min_\theta J(\theta) + \lambda \|\theta\|_2.$$

This can equivalently be phrased as

$$\begin{array}{llllll}
\text{minimize} & \sum_i t_i + \lambda r & & & & \\
\text{subject to} & t_i & \geq -\log(h_\theta(x)) & = \log(1+\exp(-\theta^T x_i)) & \text{if } y_i = 1, & \\
& t_i & \geq -\log(1-h_\theta(x)) & = \log(1+\exp(\theta^T x_i)) & \text{if } y_i = 0, & \\
& r & \geq \|\theta\|_2. & & &
\end{array} \qquad (11.20)$$

Implementation

As can be seen from (11.20) the key point is to implement the softplus bound $t \geq \log(1+e^u)$, which is the simplest example of a log-sum-exp constraint for two scalar variables t, u. This is equivalent to

$$\exp(u-t) + \exp(-t) \leq 1$$

and further to

$$\begin{array}{rcll}
(z_1, 1, u-t) & \in & K_{\exp} & (z_1 \geq \exp(u-t)), \\
(z_2, 1, -t) & \in & K_{\exp} & (z_2 \geq \exp(-t)), \\
z_1 + z_2 & \leq & 1. &
\end{array} \qquad (11.21)$$

Listing 11.9: Implementation of $t \geq \log(1+e^u)$ as in (11.21).

```python
# t >= log( 1 + exp(u) ) coordinatewise
def softplus(M, t, u):
    n = t.getShape()[0]
    z1 = M.variable(n)
    z2 = M.variable(n)
    M.constraint(Expr.add(z1, z2), Domain.equalsTo(1))
    M.constraint(Expr.hstack(z1, Expr.constTerm(n, 1.0), Expr.sub(u,t)), Domain.inPExpCone())
    M.constraint(Expr.hstack(z2, Expr.constTerm(n, 1.0), Expr.neg(t)), Domain.inPExpCone())
```

Once we have this subroutine, it is easy to implement a function that builds the regularized loss function model (11.20).

Listing 11.10: Implementation of (11.20).

```python
# Model logistic regression (regularized with full 2-norm of theta)
# X - n x d matrix of data points
# y - length n vector classifying training points
# lamb - regularization parameter
def logisticRegression(X, y, lamb=1.0):
    n, d = int(X.shape[0]), int(X.shape[1])         # num samples, dimension
    M = Model()
    theta = M.variable(d)
    t     = M.variable(n)
    reg   = M.variable()

    M.objective(ObjectiveSense.Minimize, Expr.add(Expr.sum(t), Expr.mul(lamb,reg)))
    M.constraint(Var.vstack(reg, theta), Domain.inQCone())

    signs = list(map(lambda y: -1.0 if y==1 else 1.0, y))
    softplus(M, t, Expr.mulElm(Expr.mul(X, theta), signs))

    return M, theta
```

Example: 2D dataset fitting

In the next figure we apply logistic regression to the training set of 2D points taken from the example ex2data2.txt . The two-dimensional dataset was converted into a feature vector $x \in \mathbb{R}^{28}$ using monomial coordinates of degrees at most 6.

Fig. 11.6: Logistic regression example with none, medium and strong regularization (small, medium, large λ). Without regularization we get obvious overfitting.

11.6 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. They are defined as, respectively, the largest volume ellipsoid contained inside the polytope and the smallest volume ellipsoid containing the polytope, as seen in Fig. 11.7.

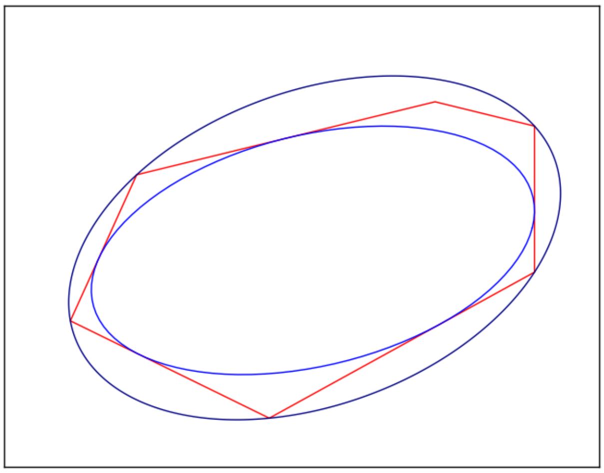


Fig. 11.7: The inner and outer Löwner-John ellipse of a polygon.

For further mathematical details, such as uniqueness of the two ellipsoids, consult *[BenTalN01]*. Our solution is a mix of conic quadratic and semidefinite programming. Among other things, in Sec. 11.6.3 we show how to implement bounds involving the determinant of a PSD matrix.

11.6.1 Inner Löwner-John Ellipsoids

Suppose we have a polytope given by an h-representation

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

and we wish to find the inscribed ellipsoid with maximal volume. It will be convenient to parametrize the ellipsoid as an affine transformation of the standard disk:

$$\mathcal{E} = \{x \mid x = Cu + d,\ u \in \mathbb{R}^n,\ \|u\|_2 \leq 1\}.$$

Every non-degenerate ellipsoid has a parametrization such that C is a positive definite symmetric $n \times n$ matrix. Now the volume of $\mathcal{E}$ is proportional to $\det(C)^{1/n}$. The condition $\mathcal{E} \subseteq \mathcal{P}$ is equivalent to the inequality $A(Cu + d) \leq b$ for all u with $\|u\|_2 \leq 1$. After a short computation we obtain the formulation:

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & t \leq \det(C)^{1/n}, \\ & (b - Ad)_i \geq \|(AC)_i\|_2, \quad i = 1, \ldots, m, \\ & C \succeq 0, \end{array} \tag{11.22}$$

where X_i denotes the i-th row of the matrix X. This can easily be implemented using *Fusion*, where the sequence of conic inequalities can be realized at once by feeding in the matrices $b - Ad$ and AC.

Listing 11.11: *Fusion* implementation of model (11.22).

```python
def lownerjohn_inner(A, b):
    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 = det_rootn(M, t, n)
        d = M.variable("d", n, Domain.unbounded())

        # (b-Ad, AC) generate cones
        M.constraint("qc", Expr.hstack(Expr.sub(b, Expr.mul(A, d)), Expr.mul(A, C)),
                     Domain.inQCone())

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

        M.solve()

        M.writeTask('lj-inner2.task')
        M.writeTask('lj-inner2.ptf')
        C, d = C.level(), d.level()
        return ([C[i:i + n] for i in range(0, n * n, n)], d)
```

The only black box is the method `det_rootn` which implements the constraint $t \leq \det(C)^{1/n}$. It will be described in Sec. 11.6.3.

11.6.2 Outer Löwner-John Ellipsoids

To compute the outer ellipsoidal approximation to a polytope, let us now start with a v-representation

$$\mathcal{P} = \text{conv}\{x_1, x_2, \ldots, x_m\} \subseteq \mathbb{R}^n,$$

of the polytope as a convex hull of a set of points. We are looking for an ellipsoid given by a quadratic inequality

$$\mathcal{E} = \{x \in \mathbb{R}^n \mid \|Px - c\|_2 \leq 1\},$$

whose volume is proportional to $\det(P)^{-1/n}$, so we are after maximizing $\det(P)^{1/n}$. Again, there is always such a representation with a symmetric, positive definite matrix P. The inclusion conditions $x_i \in \mathcal{E}$ translate into a straightforward problem formulation:

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & t \leq \det(P)^{1/n}, \\ & \|Px_i - c\|_2 \leq 1, \quad i = 1, \ldots, m, \\ & P \succeq 0, \end{array} \tag{11.23}$$

and then directly into *Fusion* code:

Listing 11.12: *Fusion* implementation of model (11.23).

```python
def lownerjohn_outer(x):
    with Model("lownerjohn_outer") as M:
        M.setLogHandler(sys.stdout)
        m, n = len(x), len(x[0])

        # Setup variables
        t = M.variable("t", 1, Domain.greaterThan(0.0))
        P = det_rootn(M, t, n)
        c = M.variable("c", n, Domain.unbounded())

        # (1, Px-c) in cone
        M.constraint("qc",
                     Expr.hstack(Expr.ones(m),
                                 Expr.sub(Expr.mul(x, P),
                                          Var.reshape(Var.repeat(c, m), [m, n])
                                          )
                                 ),
                     Domain.inQCone())

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

        M.writeTask('lj-outer2.task')
        M.writeTask('lj-outer2.ptf')
        P, c = P.level(), c.level()
        return ([P[i:i + n] for i in range(0, n * n, n)], c)
```

11.6.3 Bound on the Determinant Root

It remains to show how to express the bounds on $\det(X)^{1/n}$ for a symmetric positive definite $n \times n$ matrix X using PSD and conic quadratic variables. We want to model the set

$$C = \{(X, t) \in \mathcal{S}^n_+ \times \mathbb{R} \mid t \leq \det(X)^{1/n}\}. \tag{11.24}$$

A standard approach when working with the determinant of a PSD matrix is to consider a semidefinite cone

$$\begin{pmatrix} X & Z \\ Z^T & \mathrm{Diag}(Z) \end{pmatrix} \succeq 0 \tag{11.25}$$

where Z is a matrix of additional variables and where we intuitively identify $\mathrm{Diag}(Z) = \{\lambda_1, \ldots, \lambda_n\}$ with the eigenvalues of X. With this in mind, we are left with expressing the constraint

$$t \leq (\lambda_1 \cdot \ldots \cdot \lambda_n)^{1/n}. \tag{11.26}$$

This is easy to implement recursively using rotated quadratic cones when n is a power of 2. In general it is convenient to express (11.26) as a composition of power cones, see *[BenTalN01]* or Modeling Cookbook.

Listing 11.13: Approaching the determinant, see (11.25).

```python
def det_rootn(M, t, n):
    # Setup variables
    Y = M.variable(Domain.inPSDCone(2 * n))

    # Setup Y = [X, Z; Z^T , diag(Z)]
    X   = Y.slice([0, 0], [n, n])
```

(continues on next page)

(continued from previous page)

```
    Z   = Y.slice([0, n], [n, 2 * n])
    DZ  = Y.slice([n, n], [2 * n, 2 * n])

    # Z is lower-triangular
    M.constraint(Z.pick([[i,j] for i in range(n) for j in range(i+1,n)]), Domain.equalsTo(0.0))
    # DZ = Diag(Z)
    M.constraint(Expr.sub(DZ, Expr.mulElm(Z, Matrix.eye(n))), Domain.equalsTo(0.0))

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

    # Return an n x n PSD variable which satisfies t <= det(X)^(1/n)
    return X
```

Listing 11.14: Bounding the geometric mean, see (11.26).

```
def geometric_mean(M, x, t):
    n = int(x.getSize())
    if n==1:
      M.constraint(Expr.sub(t, x), Domain.lessThan(0.0))
    else:
      t2 = M.variable()
      M.constraint(Var.hstack(t2, x.index(n-1), t), Domain.inPPowerCone(1-1.0/n))
      geometric_mean(M, x.slice(0,n-1), t2)
```

11.7 SUDOKU

SUDOKU is a famous simple yet mind-blowing 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:

				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	*2*	*9*	*5*	4	*7*	*6*	*1*	*8*
6	5	8	*9*	*1*	3	*4*	*2*	*7*
4	1	*7*	2	8	*6*	9	*5*	*3*
9	7	3	1	*5*	*2*	8	4	*6*
5	*6*	*2*	*8*	*7*	*4*	*3*	*9*	*1*
8	4	1	*6*	*3*	9	2	7	*5*
1	*9*	4	*3*	6	5	*7*	8	*2*
7	*3*	*5*	4	*2*	*8*	1	6	*9*
2	*8*	*6*	*7*	9	*1*	*5*	*3*	*4*

The solution

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 $\{(mt, ml),\ (m(t+1)-1, m(l+1)-1)\}$, for $t, l = 0, \ldots, m-1$ (we index cells from $(0, 0)$).

In general, each SUDOKU instance comes with a set F of predetermined values which:

- reduce the complexity of the game by removing symmetries and guiding the initial moves of the player;
- ensure that there will be a unique solution.

We represent the set F as list of triplets (i,j,v), meaning that the cell (i,j) contains the value v.

Note that SUDOKU is a **feasibility** problem. A typical Integer Programming formulation is straightforward: let x_{ijk} be a binary variable that takes value 1 if k is written in cell (i,j). Then we look for a feasible solution of a system of constraints given below.

SUDOKU is a typical assignment problem. Its constraints are commonly found in optimization problems concerning scheduling or resource allocation. SUDOKU has also been a nice problem to fiddle with for many researchers in the optimization community. Indeed, its simple structure and the easy way in which the results can be tested make it a perfect test problem.

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 value 1 if k is written in the cell (i,j), or 0 otherwise. We first ask that for each cell exactly one digit is selected:

$$\sum_{k=0}^{n-1} x_{ijk} = 1, \qquad i,j = 0,\ldots,n-1. \tag{11.27}$$

Similar constraints can be used to force each digit to appear only once in each row or column:

$$\begin{array}{ll} \sum_{i=0}^{n-1} x_{ijk} = 1, & j,k = 0,\ldots,n-1, \\ \sum_{j=0}^{n-1} x_{ijk} = 1, & i,k = 0,\ldots,n-1. \end{array} \tag{11.28}$$

To force a digit to appear only once in each sub-grid we can use the following

$$\sum_{i=0}^{m-1}\sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1 \qquad k = 0,\ldots,n-1 \text{ and } t,l = 0,\ldots,m-1 \tag{11.29}$$

If a cell (i,j) has a predetermined value, i.e. $(i,j,k) \in F$ then we set

$$x_{ijk} = 1.$$

Summarizing, and considering that there is no objective function to minimize, the optimization model for the SUDOKU problem takes the form

$$\begin{array}{ll} \min 0 & \\ \text{s.t.} & \\ \sum_{i=0}^{n-1} x_{ijk} = 1, & j,k = 0,\ldots,n-1, \\ \sum_{j=0}^{n-1} x_{ijk} = 1, & i,k = 0,\ldots,n-1, \\ \sum_{k=0}^{n-1} x_{ijk} = 1, & i,j = 0,\ldots,n-1, \\ \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1, & k = 0,\ldots,n-1 \text{ and} \\ & t,l = 0,\ldots,m-1, \\ & \\ x_{ijk} = 1, & \forall (i,j,k) \in F. \end{array} \tag{11.30}$$

Implementation with *Fusion*

The implementation in *Fusion* is straightforward. First, we represent the variable x using a three dimensional *Fusion* variable:

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

Then we can define constraints (11.27) and (11.28) simply using the *Expr.sum* operator, that allows to sum the elements of an expression (in this case of the variable itself) along arbitrary dimensions. The code reads:

```
        #each value only once per dimension
        for d in range(m):
            M.constraint("row_sum(%d)" % d, Expr.sum(x, d), Domain.equalsTo(1.))
```

The last set of constraints (11.29) , 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("blocksum(%d,%d,k=%d)" % (i,j,k),
                                 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("fix",x.pick(fixed), Domain.equalsTo(1.0))
```

SUDOKU: the complete example code.

The complete code for the SUDOKU problem is shown in Listing 11.15.

Listing 11.15: *Fusion* implementation to solve SUDOKU.

```
import sys
import mosek
from mosek.fusion import *
import numpy as npy


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],
```

(continues on next page)

(continued from previous page)

```
                [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("x",[n, n, n], Domain.binary())

        #each value only once per dimension
        for d in range(m):
            M.constraint("row_sum(%d)" % d, 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("blocksum(%d,%d,k=%d)" % (i,j,k),
                                 Expr.sum(x.slice([i * m, j * m, k], [(i + 1) * m, (j + 1) * m,
↪ k + 1])),
                                 Domain.equalsTo(1.))

        M.constraint("fix",x.pick(fixed), Domain.equalsTo(1.0))

        M.setLogHandler(sys.stdout)

        M.solve()

        M.writeTask("sudoku.task")

        #print the solution, if any...
        if M.getPrimalSolutionStatus() in [SolutionStatus.Optimal]:
            print_solution(m, x)
        else:
            print("No solution found!")

if __name__ == '__main__':
    main()
```

The problem instance corresponding to Fig. 11.7 is hard-coded for the sake of simplicity. It will produce the following output

```
Problem
  Name                    : SUDOKU
  Objective sense         : min
  Type                    : LO (linear optimization problem)
  Constraints             : 350
  Cones                   : 0
  Scalar variables        : 1000
  Matrix variables        : 0
  Integer variables       : 729

Optimizer started.
Mixed integer optimizer started.
Threads used: 2
Presolve started.
Presolve terminated. Time = 0.00
Presolved problem: 0 variables, 0 constraints, 0 non-zeros
Presolved problem: 0 general integer, 0 binary, 0 continuous
```

(continues on next page)

(continued from previous page)

```
Clique table size: 0
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(%)  TIME
0        1        0        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           : 0
Number of branches                 : 0
Number of relaxations solved       : 1
Number of interior point iterations: 0
Number of simplex iterations       : 0
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   : 0.00
Mixed integer optimizer terminated. Time: 0.02

Optimizer terminated. Time: 0.02


 | 3 2 9 | 5 4 7 | 6 1 8 |
 | 6 5 8 | 9 1 3 | 4 2 7 |
 | 4 1 7 | 2 8 6 | 9 5 3 |


 | 9 7 3 | 1 5 2 | 8 4 6 |
 | 5 6 2 | 8 7 4 | 3 9 1 |
 | 8 4 1 | 6 3 9 | 2 7 5 |


 | 1 9 4 | 3 6 5 | 7 8 2 |
 | 7 3 5 | 4 2 8 | 1 6 9 |
 | 2 8 6 | 7 9 1 | 5 3 4 |
```

11.8 Travelling Salesman Problem (TSP)

The *Travelling Salesman Problem* is one of the most famous and studied problems in combinatorics and integer optimization. In this case study we shall:

- 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 adding more constraints;
- show how to access the solution of an optimization problem.

The material presented in this section draws inspiration from *[Pat03]*.

In a TSP instance 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}. The goal is to find a minimum cost *Hamilton cycle* in G, that is a closed tour passing through each node exactly once. For example, consider the small directed graph in Fig. 11.8.

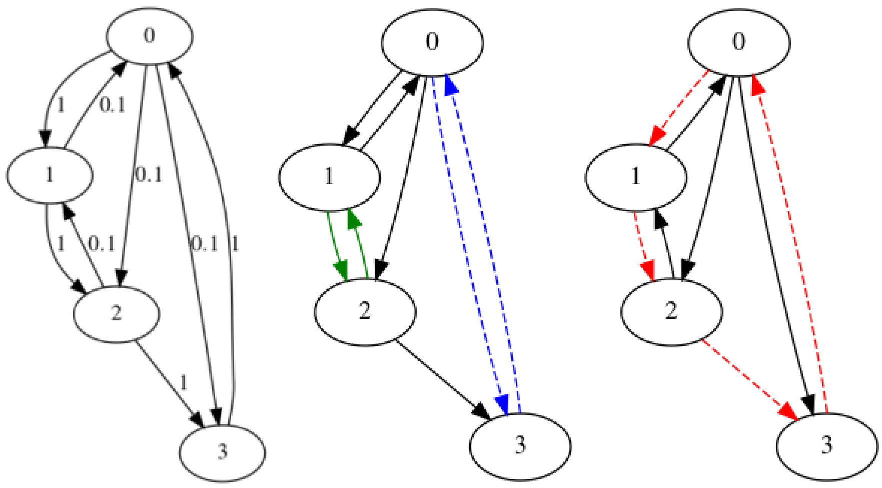


Fig. 11.8: (Left) a directed graph with costs. (Middle) The minimum cycle cover found in the first iteration. (Right) The minimum cost travelling salesman tour.

Its corresponding adjacency and cost matrices A and c are:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad c = \begin{bmatrix} 0 & 1 & 0.1 & 0.1 \\ 0.1 & 0 & 1 & 0 \\ 0 & 0.1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Typically, the problem is modeled introducing a set of binary variables x_{ij} such that

$$x_{ij} = \begin{cases} 0 & \text{if arc } (i,j) \text{ is in the tour,} \\ 1 & \text{otherwise.} \end{cases}$$

Now we can introduce the following simple model:

$$\begin{array}{lll} \min & \sum_{i,j} c_{ij} x_{ij} & \\ \text{subject to} & \sum_i x_{ij} = 1 & \forall j = 1, \ldots, n, \\ & \sum_j x_{ij} = 1 & \forall i = 1, \ldots, n, \\ & x_{ij} \leq A_{ij} & \forall i, j, \\ & x_{ij} \in \{0, 1\} & \forall i, j. \end{array} \tag{11.31}$$

It describes the constraint that every vertex has exactly one incoming and one outgoing arc in the tour, and that only arcs present in the graph can be chosen. Problem (11.31) 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.constraint(x, Domain.lessThan( A ))

    M.objective(ObjectiveSense.Minimize, Expr.dot(C ,x))
```

Note in particular how:

- we can sum over rows and/or columns using the *Expr.sum* function;
- we use *Expr.dot* to compute the objective function.

The solution to problem (11.31) is not necessarily a closed tour. In fact (11.31) models another problem known as *minimum cost cycle cover*, whose solution may consist of more than one cycle. In our example we get the solution depicted in Fig. 11.8, i.e. there are two loops, namely *0->3->0* and *1->2->1*.

A solution to (11.31) solves the TSP problem if and only if it consists of a single cycle. One classical approach ensuring this is the so-called *subtour elimination*: once we found a solution of (11.31) composed of at least two cycles, we add constraints that explicitly avoid that particular solution:

$$\sum_{(i,j)\in c} x_{ij} \leq |c| - 1 \quad \forall c \in C. \tag{11.32}$$

Thus the problem we want to solve at each step is

$$\begin{array}{lll} \min & \sum_{i,j} c_{ij}x_{ij} & \\ \text{subject to} & \sum_i x_{ij} = 1 & \forall j = 1,\dots,n, \\ & \sum_j x_{ij} = 1 & \forall i = 1,\dots,n, \\ & x_{ij} \leq A_{ij} & \forall i,j, \\ & x_{ij} \in \{0,1\} & \forall i,j, \\ & \sum_{(i,j)\in c} x_{ij} \leq |c| - 1 & \forall c \in C, \end{array} \tag{11.33}$$

where C is the set of cycles in all the cycle covers we have seen so far. The overall solution scheme is the following:

1. set C as the empty set,
2. solve problem (11.33),
3. **if** x has only one cycle **stop**,
4. **else** add the cycles of x to C and **goto** 2.

Cycle detection is a fairly easy task and we omit the procedure here for the sake of simplicity. Now we show how to add a constraint for each cycle. Since we have the list of arcs, and each one corresponds to a variable x_{ij}, we can use the function *Variable.pick* to compactly define constraints of the form (11.32):

```
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; then the second iteration is forced not to include those cycles, and a new solution is located. This time it consists of one loop, and as expected the cost is higher. The solution is depicted in Fig. 11.8.

Formulation (11.33) can be improved in some cases by exploiting the graph structure. Some simple tricks follow.

Self-loops

Self-loops are never part of a TSP tour. Typically self-loops are removed by penalizing them with a huge cost c_{ii}. Although this works in practice, it is more advisable to just fix the corresponding variables to zero, i.e.

$$x_{ii} = 0 \quad \forall i = 1, \ldots, n. \tag{11.34}$$

This removes redundant variables, and avoids unnecessarily large coefficients that can negatively affect the solver.

Constraints (11.34) are easily implemented as follows:

```
M.constraint(x.diag(), Domain.equalsTo(0.))
```

Two-arc loops removal

In networks with more than two nodes two-loop arcs can also be ignored. They are simple to detect and their number is of the same order as the size of the graph. The constraints we need to add are:

$$x_{ij} + x_{ji} \leq 1 \quad \forall i, j = 1, \ldots, n. \tag{11.35}$$

Constraints (11.35) are easily implemented as follows:

```
M.constraint(Expr.add(x, x.transpose()), Domain.lessThan(1.0))
```

The complete working example

Listing 11.16: The complete code for the TSP examples.

```
def tsp(n, A, C, 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.constraint(x, Domain.lessThan( A ))

        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 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])
```

(continues on next page)

(continued from previous page)

```
            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)
    x,c = tsp(n, Matrix.sparse(n,n,A_i,A_j,1.), costs , True, False)
```

11.9 Nearest Correlation Matrix Problem

A *correlation matrix* is a symmetric positive definite matrix with unit diagonal. This term has origins in statistics, since the matrix whose entries are the correlation coefficients of a sequence of random variables has all these properties.

In this section we study variants of the problem of approximating a given symmetric matrix A with correlation matrices:

- find the correlation matrix X nearest to A in the *Frobenius norm*,
- find an approximation of the form $D + X$ where D is a diagonal matrix with positive diagonal and X is a positive semidefinite matrix of low rank, using the combination of Frobenius and *nuclear norm*.

Both problems are related to *portfolio optimization*, where one can often have a matrix A that only approximates the correlations of stocks. For subsequent optimizations one would like to approximate A with a correlation matrix or, in the factor model, with $D + VV^T$ with VV^T of small rank.

11.9.1 Nearest correlation with the Frobenius norm

The Frobenius norm of a real matrix M is defined as

$$\|M\|_F = \left(\sum_{i,j} M_{i,j}^2\right)^{1/2}$$

and with respect to this norm our optimization problem can be expressed simply as:

$$\begin{array}{ll} \text{minimize} & \|A - X\|_F \\ \text{subject to} & \mathbf{diag}(X) = e, \\ & X \succeq 0. \end{array} \tag{11.36}$$

We can exploit the symmetry of A and X to get a compact vector representation. To this end we make use of the following mapping from a symmetric matrix to a flattened vector containing the (scaled) lower triangular part of the matrix:

$$\begin{array}{ll} \text{vec}: & \mathbb{R}^{n\times n} \to \mathbb{R}^{n(n+1)/2} \\ \text{vec}(M) = & (\alpha_{11}M_{11}, \alpha_{21}M_{21}, \alpha_{22}M_{22}, \ldots, \alpha_{n1}M_{n1}, \ldots, \alpha_{nn}M_{nn}) \\ \alpha_{ij} = & \begin{cases} 1 & j = i \\ \sqrt{2} & j < i \end{cases} \end{array} \tag{11.37}$$

Note that $\|M\|_F = \|\text{vec}(M)\|_2$. The *Fusion* implementation of vec is as follows:

Listing 11.17: Implementation of function *vec* in (11.37).

```
def vec(e):
    N = e.getShape()[0]

    msubi = range(N * (N + 1) // 2)
    msubj = [i * N + j for i in range(N) for j in range(i + 1)]
    mcof  = [2.0**0.5 if i !=
             j else 1.0 for i in range(N) for j in range(i + 1)]

    S = Matrix.sparse(N * (N + 1) // 2, N * N, msubi, msubj, mcof)
    return Expr.mul(S, Expr.flatten(e))
```

That leads to an optimization problem with both conic quadratic and semidefinite constraints:

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & (t, \text{vec}(A - X)) \in \mathcal{Q}, \\ & \mathbf{diag}(X) = e, \\ & X \succeq 0. \end{array} \tag{11.38}$$

Code example

Listing 11.18: Implementation of problem (11.38).

```
def nearestcorr(A):
    N = A.numRows()

    # Create a model
    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())
```

(continues on next page)

(continued from previous page)

```
        # diag(X) = e
        M.constraint("C2", X.diag(), Domain.equalsTo(1.0))

        # Objective: Minimize t
        M.objective(ObjectiveSense.Minimize, t)
        M.writeTask('nearcor.task')
        M.writeTask('nearcor.cbf')
        M.solve()

        return X.level(), t.level()
```

We use the following input

Listing 11.19: 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 output 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.        ]]
```

11.9.2 Nearest Correlation with Nuclear-norm Penalty

Next, we consider the approximation of A of the form $D + X$ where $D = \mathbf{diag}(w)$, $w \geq 0$ and $X \succeq 0$. We will also aim at minimizing the rank of X. This can be approximated by a relaxed linear objective penalizing the trace $\mathrm{Tr}(X)$ (which in this case is the *nuclear norm* of X and happens to be the sum of its eigenvalues).

The combination of these constraints leads to a problem:

$$\begin{array}{ll} \text{minimize} & \|X + \mathbf{diag}(w) - A\|_F + \gamma \mathrm{Tr}(X), \\ \text{subject to} & X \succeq 0, w \geq 0, \end{array}$$

where the parameter γ controls the tradeoff between the quality of approximation and the rank of X.

Exploit the mapping vec defined in (11.37) we can express this problem as:

$$\begin{array}{ll} \text{minimize} & t + \gamma \mathrm{Tr}(X) \\ \text{subject to} & (t, \mathrm{vec}(X + \mathbf{diag}(w) - A)) \in \mathcal{Q}, \\ & X \succeq 0, w \geq 0. \end{array} \tag{11.39}$$

Code example

Listing 11.20: Implementation of problem (11.39).

```
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", Domain.inPSDCone(N))
```

(continues on next page)

```
    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())

    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()

        # Find eigenvalues of X and compute its rank
        d = [0.0] * int(N)
        LinAlg.syeig(mosek.uplo.lo, N, X.level(), d)
        result.append(
            (g, t.level(), sum([d[i] > 1e-6 for i in range(N)]), X.level()))

    return result
```

We feed **MOSEK** with the same input as in Sec. 11.9.1. The problem is solved for a range of values γ values, to demonstrate how the penalty term helps achieve a low rank solution. To this extent we report both the rank of X and the residual norm $\|X + \mathbf{diag}(w) - A\|_F$.

```
--- 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
```

11.10 Semidefinite Relaxation of MIQCQO Problems

In this case study we will discuss a fairly common application for Semidefinite Optimization: to define a continuous semidefinite relaxation of a mixed-integer quadratic optimization problem. This section is based on the method by Park and Boyd *[PB15]*.

We will focus on problems of the form:

$$\begin{array}{ll} \text{minimize} & x^T P x + 2q^T x \\ \text{subject to} & x \in \mathbb{Z}^n \end{array} \tag{11.40}$$

where $q \in \mathbb{R}^n$ and $P \in \mathcal{S}_+^{n\times n}$ is positive semidefinite. There are many important problems that can be reformulated as (11.40), for example:

- *integer least squares*: minimize $\|Ax - b\|_2^2$ subject to $x \in \mathbb{Z}^n$,
- *closest vector problem*: minimize $\|v - z\|_2$ subject to $z \in \{Bx \mid x \in \mathbb{Z}^n\}$.

Following *[PB15]*, we can derive a relaxed continuous model. We first relax the integrality constraint

$$\begin{array}{ll} \text{minimize} & x^T P x + 2q^T x \\ \text{subject to} & x_i(x_i - 1) \geq 0 \quad i = 1, \dots, n. \end{array}$$

The last constraint is still non-convex. We introduce a new variable $X \in \mathbb{R}^{n\times n}$, such that $X = x \cdot x^T$. This allows us to write an equivalent formulation:

$$\begin{array}{ll} \text{minimize} & \mathrm{Tr}(PX) + 2q^T x \\ \text{subject to} & \mathbf{diag}(X) \geq x, \\ & X = x \cdot x^T. \end{array}$$

To get a conic problem we relax the last constraint and apply the Schur complement. The final relaxation follows:

$$\begin{array}{ll} \text{minimize} & \mathrm{Tr}(PX) + 2q^T x \\ \text{subject to} & \mathbf{diag}(X) \geq x, \\ & \left[\begin{array}{cc} X & x \\ x^T & 1 \end{array}\right] \in \mathcal{S}_+^{n+1}. \end{array} \tag{11.41}$$

Fusion Implementation

Implementing model (11.41) in *Fusion* is very simple. We assume the input n, P and q. Then we proceed creating the optimization model

```
M = Model()
```

The important step is to define a single PSD variable

$$Z = \left[\begin{array}{cc} X & x \\ x^T & 1 \end{array}\right] \in \mathcal{S}_+^{n+1}.$$

Our code will create Z and two slices that correspond to X and x:

```
Z = M.variable("Z", Domain.inPSDCone(n+1))
X = Z.slice([0,0], [n,n])
x = Z.slice([0,n], [n,n+1])
```

Then we define the constraints:

```
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( P, X ) ),
    Expr.mul( 2.0, Expr.dot(x, q) )
) )
```

Note that the *trace* operator is not directly available in *Fusion*, but it can easily be defined from scratch.

Complete code

Listing 11.21: *Fusion* implementation of model (11.41).

```
def miqcqp_sdo_relaxation(n,P,q):
    M = Model()

    M.setLogHandler(sys.stdout)

    Z = M.variable("Z", Domain.inPSDCone(n+1))
    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.) )
```

(continues on next page)

(continued from previous page)

```
    M.objective( ObjectiveSense.Minimize, Expr.add(
        Expr.sum( Expr.mulElm( P, X ) ),
        Expr.mul( 2.0, Expr.dot(x, q) )
    ) )
    return M
```

Numerical Examples

We present now some simple numerical experiments for the integer least squares problem:

$$\begin{array}{ll} \text{minimize} & \|Ax-b\|_2^2 \\ \text{subject to} & x \in \mathbb{Z}^n. \end{array} \tag{11.42}$$

It corresponds to the problem (11.40) with $P = A^T A$ and $q = -A^T b$. Following *[PB15]* we will generate the input data by taking all entries of A from the normal distribution $\mathcal{N}(0,1)$ and setting $b = Ac$ where c comes from the uniform distribution on $[0,1]$.

An integer rounding `xRound` of the solution to (11.41) is a feasible integer solution to (11.42). We can compare it to the actual optimal integer solution `xOpt`, whenever the latter is available. Of course it is very simple to formulate the integer least squares problem in *Fusion*:

```
def int_least_squares(n, A, b):
    M = Model()

    M.setLogHandler(sys.stdout)

    x = M.variable("x", n, Domain.integral(Domain.unbounded()))
    t = M.variable("t", 1, Domain.unbounded())

    M.constraint( Expr.vstack(t, Expr.sub(Expr.mul(A, x), b)), Domain.inQCone() )
    M.objective( ObjectiveSense.Minimize, t )

    return M
```

All that remains is to compare the values of the objective function $\|Ax - b\|_2$ for the two solutions.

Listing 11.22: The comparison of two solutions.

```
# problem dimensions
n = 20
m = 2*n

# problem data
A = numpy.reshape(numpy.random.normal(0., 1.0, n*m), (m,n))
c = numpy.random.uniform(0., 1.0, n)
P = A.transpose().dot(A)
q = - P.dot(c)
b = A.dot(c)

# solve the problems
M = miqcqp_sdo_relaxation(n, P, q)
Mint = int_least_squares(n, A, b)
M.solve()
Mint.solve()

M.writeTask('M.ptf')
M.writeTask('Mint.ptf')

# rounded and optimal solution
xRound = numpy.rint(M.getVariable("Z").slice([0,n], [n,n+1]).level())
xOpt = numpy.rint(Mint.getVariable("x").level())
```

(continues on next page)

(continued from previous page)

```
print(M.getSolverDoubleInfo("optimizerTime"), Mint.getSolverDoubleInfo("optimizerTime"))
print(numpy.linalg.norm(A.dot(xRound)-b), numpy.linalg.norm(A.dot(xOpt)-b))
```

Experimentally the objective value for `xRound` approximates the optimal solution with a factor of 1.1-1.4. We refer to *[PB15]* for a more involved iterative rounding procedure, producing integer solutions of even better quality, and for a detailed discussion of test results.

Chapter 12

Problem Formulation and Solutions

In this chapter we will discuss the following issues:

- The formal, mathematical formulations of the problem types that **MOSEK** can solve and their duals.
- The solution information produced by **MOSEK**.
- The infeasibility certificate produced by **MOSEK** if the problem is infeasible.

For the underlying mathematical concepts, derivations and proofs see the Modeling Cookbook or any book on convex optimization. This chapter explains how the related data is organized specifically within the **MOSEK** API.

12.1 Linear Optimization

MOSEK accepts linear optimization problems of the form

$$\begin{array}{lccccc} \mbox{minimize} & & & c^T x + c^f & & \\ \mbox{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \end{array} \tag{12.1}$$

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.
- $c^f \in \mathbb{R}$ is a constant term in the objective
- $A \in \mathbb{R}^{m \times n}$ 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.

Lower and upper bounds can be infinite, or in other words the corresponding bound may be omitted.

A primal solution (x) is *(primal) feasible* if it satisfies all constraints in (12.1). If (12.1) has at least one primal feasible solution, then (12.1) is said to be (primal) feasible. In case (12.1) does not have a feasible solution, the problem is said to be *(primal) infeasible*

12.1.1 Duality for Linear Optimization

Corresponding to the primal problem (12.1), there is a dual problem

$$
\begin{array}{lccl}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f & & \\
& A^T y + s_l^x - s_u^x & = & c, \\
\text{subject to} & -y + s_l^c - s_u^c & = & 0, \\
& s_l^c, s_u^c, s_l^x, s_u^x & \geq & 0.
\end{array}
\tag{12.2}
$$

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. This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. In other words:

$$
l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0 \text{ and } l_j^x \cdot (s_l^x)_j = 0.
$$

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 (12.2). If (12.2) has at least one feasible solution, then (12.2) is *(dual) feasible*, otherwise the problem is *(dual) infeasible*.

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 (12.1) and $(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$ is a solution to the corresponding dual problem (12.2). We also define an auxiliary vector

$$
(x^c)^* := Ax^*
$$

containing the activities of linear constraints.

For a primal-dual feasible solution we define the *duality gap* as the difference between the primal and the dual objective value,

$$
\begin{array}{l}
c^T x^* + c^f - \left\{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \right\} \\
= \sum_{i=0}^{m-1} \left[(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*) \right] \\
+ \sum_{j=0}^{n-1} \left[(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*) \right] \geq 0
\end{array}
\tag{12.3}
$$

where the first relation can be obtained by transposing and multiplying the dual constraints (12.2) 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.

It is well-known that a linear optimization problem has an optimal solution if and only if there exist feasible primal-dual solution 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 (12.1) 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.

12.1.2 Infeasibility for Linear Optimization

Primal Infeasible Problems

If the problem (12.1) 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

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\ \text{subject to} & \\ & A^T y + s_l^x - s_u^x = 0, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \end{array} \tag{12.4}$$

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 (12.4) 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 (12.4) is unbounded, and that (12.1) is infeasible.

Dual Infeasible Problems

If the problem (12.2) 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

$$\begin{array}{lccccl} \text{minimize} & & & c^T x & & \\ \text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\ & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \end{array} \tag{12.5}$$

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 (12.5) is unbounded, and that (12.2) is infeasible.

In case that both the primal problem (12.1) and the dual problem (12.2) are infeasible, **MOSEK** will report only one of the two possible certificates — which one is not defined (**MOSEK** returns the first certificate found).

12.1.3 Minimalization vs. Maximalization

When the objective sense of problem (12.1) is maximization, i.e.

$$\begin{array}{lccccl} \text{maximize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \end{array}$$

the objective sense of the dual problem changes to minimization, and the domain of all dual variables changes sign in comparison to (12.2). 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 (12.3) 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

$$\begin{array}{c} A^T y + s_l^x - s_u^x = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \end{array} \tag{12.6}$$

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 (12.5) such that $c^T x > 0$.

12.2 Conic Optimization

Conic optimization is an extension of linear optimization (see Sec. 12.1) allowing conic domains to be specified for subsets of the problem variables. A conic optimization problem to be solved by **MOSEK** can be written as

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & x \in \mathcal{K}, & & \end{array} \tag{12.7}$$

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.
- $c^f \in \mathbb{R}$ is a constant term in the objective
- $A \in \mathbb{R}^{m \times n}$ 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.

Lower and upper bounds can be infinite, or in other words the corresponding bound may be omitted.

The 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. The user only needs to specify subsets of variables which belong to non-trivial cones.

In this section we discuss the formulations which apply to the following cones supported by **MOSEK**:

- The set $\mathbb{R}^n$.
- The zero cone $\{(0, \ldots, 0)\}$.
- Quadratic cone

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n : x_1 \geq \sqrt{\sum_{j=2}^{n} x_j^2} \right\}.$$

- Rotated quadratic cone

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n : 2x_1x_2 \geq \sum_{j=3}^{n} x_j^2, \quad x_1 \geq 0, \quad x_2 \geq 0 \right\}.$$

- Primal exponential cone

$$K_{\exp} = \left\{ x \in \mathbb{R}^3 : x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0 \right\}$$

as well as its dual

$$K_{\exp}^* = \left\{ x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0 \right\}.$$

- Primal power cone (with parameter $0 < \alpha < 1$)

$$\mathcal{P}_n^{\alpha,1-\alpha} = \left\{ x \in \mathbb{R}^n : x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n} x_j^2}, \quad x_1, x_2 \geq 0 \right\}$$

as well as its dual

$$(\mathcal{P}_n^{\alpha,1-\alpha})^* = \left\{ x \in \mathbb{R}^n : \left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n} x_j^2}, \quad x_1, x_2 \geq 0 \right\}.$$

MOSEK supports also the cone of positive semidefinite matrices. Since that is handled through a separate interface, we discuss it in Sec. 12.3.

12.2.1 Duality for Conic Optimization

Corresponding to the primal problem (12.7), there is a dual problem

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ \text{subject to} & \\ & A^T y + s_l^x - s_u^x + s_n^x = c \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ & s_n^x \in \mathcal{K}^*, \end{array} \tag{12.8}$$

where the dual cone $\mathcal{K}^*$ is a Cartesian product of the cones dual to $\mathcal{K}_t$. In practice this means that s_n^x has one entry for each entry in x. Please note that the dual problem of the dual problem is identical to the original primal problem.

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. This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. In other words:

$$l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0 \text{ and } l_j^x \cdot (s_l^x)_j = 0.$$

A solution

$$(y, s_l^c, s_u^c, s_l^x, s_u^x, s_n^x)$$

to the dual problem is feasible if it satisfies all the constraints in (12.8). If (12.8) has at least one feasible solution, then (12.8) is *(dual) feasible*, otherwise the problem is *(dual) infeasible*.

A solution

$$(x^*, y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$$

is denoted a *primal-dual feasible solution*, if (x^*) is a solution to the primal problem (12.7) and $(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$ is a solution to the corresponding dual problem (12.8). We also define an auxiliary vector

$$(x^c)^* := Ax^*$$

containing the activities of linear constraints.

For a primal-dual feasible solution we define the *duality gap* as the difference between the primal and the dual objective value,

$$\begin{array}{l} c^T x^* + c^f - \left\{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \right\} \\ = \sum_{i=0}^{m-1} \left[(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*) \right] \\ + \sum_{j=0}^{n-1} \left[(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*) \right] + \sum_{j=0}^{n-1} (s_n^x)_j^* x_j^* \geq 0 \end{array} \tag{12.9}$$

where the first relation can be obtained by transposing and multiplying the dual constraints (12.2) 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.

It is well-known that, under some non-degeneracy assumptions that exclude ill-posed cases, a conic optimization problem has an optimal solution if and only if there exist feasible primal-dual solution so that the duality gap is zero, or, equivalently, that the *complementarity conditions*

$$\begin{array}{rcll} (s_l^c)_i^* ((x_i^c)^* - l_i^c) & = & 0, & i = 0, \ldots, m-1, \\ (s_u^c)_i^* (u_i^c - (x_i^c)^*) & = & 0, & i = 0, \ldots, m-1, \\ (s_l^x)_j^* (x_j^* - l_j^x) & = & 0, & j = 0, \ldots, n-1, \\ (s_u^x)_j^* (u_j^x - x_j^*) & = & 0, & j = 0, \ldots, n-1, \\ \sum_{j=0}^{n-1} (s_n^x)_j^* x_j^* & = & 0. & \end{array} \tag{12.10}$$

are satisfied.

If (12.7) 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.

12.2.2 Infeasibility for Conic Optimization

Primal Infeasible Problems

If the problem (12.7) 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

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\ \text{subject to} & \\ & \begin{array}{c} A^T y + s_l^x - s_u^x + s_n^x = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ s_n^x \in \mathcal{K}^*, \end{array} \end{array} \tag{12.11}$$

such that the objective value is strictly positive, i.e. a solution

$$(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$$

to (12.11) 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 (12.11) is unbounded, and that (12.7) is infeasible.

Dual Infeasible Problems

If the problem (12.8) 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

$$\begin{array}{lccccc} \text{minimize} & & & c^T x & & \\ \text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\ & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \\ & & & x \in K, & & \end{array} \tag{12.12}$$

where

$$\hat{l}_i^c = \left\{ \begin{array}{ll} 0 & \text{if } l_i^c > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_i^c := \left\{ \begin{array}{ll} 0 & \text{if } u_i^c < \infty, \\ \infty & \text{otherwise,} \end{array} \right\} \tag{12.13}$$

and

$$\hat{l}_j^x = \left\{ \begin{array}{ll} 0 & \text{if } l_j^x > -\infty, \\ -\infty & \text{otherwise,} \end{array} \right\} \quad \text{and} \quad \hat{u}_j^x := \left\{ \begin{array}{ll} 0 & \text{if } u_j^x < \infty, \\ \infty & \text{otherwise,} \end{array} \right\} \tag{12.14}$$

such that

$$c^T x < 0.$$

Such a solution implies that (12.12) is unbounded, and that (12.8) is infeasible.

In case that both the primal problem (12.7) and the dual problem (12.8) are infeasible, **MOSEK** will report only one of the two possible certificates — which one is not defined (**MOSEK** returns the first certificate found).

12.2.3 Minimalization vs. Maximalization

When the objective sense of problem (12.7) is maximization, i.e.

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

the objective sense of the dual problem changes to minimization, and the domain of all dual variables changes sign in comparison to (12.2). The dual problem thus takes the form

$$\begin{array}{ll} \text{minimize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ \text{subject to} & A^T y + s_l^x - s_u^x + s_n^x = c, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ & -s_n^x \in \mathcal{K}^* \end{array}$$

This means that the duality gap, defined in (12.9) 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

$$\begin{array}{c} A^T y + s_l^x - s_u^x + s_n^x = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ -s_n^x \in \mathcal{K}^* \end{array} \tag{12.15}$$

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 (12.12) such that $c^T x > 0$.

12.3 Semidefinite Optimization

Semidefinite optimization is an extension of conic optimization (see Sec. 12.2) allowing positive semidefinite matrix variables to be used in addition to the usual scalar variables. All the other parts of the input are defined exactly as in Sec. 12.2, and the discussion from that section applies verbatim to all properties of problems with semidefinite variables. We only briefly indicate how the corresponding formulae should be modified with semidefinite terms.

A semidefinite optimization problem can be written as

$$
\begin{array}{lccccl}
\text{minimize} & & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle + c^f & & \\
\text{subject to} & l_i^c & \leq & \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \leq & u_i^c, \quad i = 0, \ldots, m-1 \\
 & l_j^x & \leq & x_j & \leq & u_j^x, \quad j = 0, \ldots, n-1 \\
 & & & x \in \mathcal{K}, & & \\
 & & & \overline{X}_j \in \mathcal{S}_+^{r_j}, & & \quad j = 0, \ldots, p-1
\end{array}
\tag{12.16}
$$

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

$$
\langle U, V \rangle := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} U_{ij} V_{ij}.
$$

As always we write $A = (a_{i,j})$ for the linear coefficient matrix.

Duality

The definition of the dual problem (12.8) becomes:

$$
\begin{array}{lll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f & \\
\text{subject to} & & \\
 & A^T y + s_l^x - s_u^x + s_n^x = c & \\
 & -y + s_l^c - s_u^c = 0, & \\
 & \overline{C}_j - \sum_{i=0}^{m-1} y_i \overline{A}_{ij} = \overline{S}_j, & j = 0, \ldots, p-1 \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\
 & s_n^x \in \mathcal{K}^*, & \\
 & \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 0, \ldots, p-1.
\end{array}
\tag{12.17}
$$

The duality gap (12.9) is computed as:

$$
\begin{array}{l}
c^T x^* + c^f - \left\{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \right\} \\
= \sum_{i=0}^{m-1} \left[(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*) \right] \\
+ \sum_{j=0}^{n-1} \left[(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*) \right] + \sum_{j=0}^{n-1} (s_n^x)_j^* x_j^* + \sum_{j=0}^{p-1} \langle \overline{X}_j, \overline{S}_j \rangle \geq 0.
\end{array}
\tag{12.18}
$$

Complementarity conditions (12.10) include the additional relation:

$$
\langle \overline{X}_j, \overline{S}_j \rangle = 0 \quad j = 0, \ldots, p-1.
\tag{12.19}
$$

Infeasibility

A certificate of primal infeasibility (12.11) is now a feasible solution to:

$$
\begin{array}{lll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x & \\
\text{subject to} & & \\
 & A^T y + s_l^x - s_u^x + s_n^x = 0, & \\
 & -y + s_l^c - s_u^c = 0, & \\
 & \sum_{i=0}^{m-1} y_i \overline{A}_{ij} + \overline{S}_j = 0, & j = 0, \ldots, p-1 \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\
 & s_n^x \in \mathcal{K}^*, & \\
 & \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 0, \ldots, p-1.
\end{array}
\tag{12.20}
$$

such that the objective value is strictly positive.

Similarly, a dual infeasibility certificate (12.12) is a feasible solution to

$$
\begin{array}{lccccl}
\text{minimize} & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle & & & \\
\text{subject to} & \hat{l}_i^c \leq & \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \leq & \hat{u}_i^c, & i = 0, \ldots, m-1 \\
& \hat{l}_j^x \leq & x_j & \leq & \hat{u}_j^x, & j = 0, \ldots, n-1 \\
& & x \in \mathcal{K}, & & & \\
& & \overline{X}_j \in \mathcal{S}_+^{r_j}, & & & j = 0, \ldots, p-1
\end{array}
\tag{12.21}
$$

where the modified bounds are as in (12.13) and (12.14) and the objective value is strictly negative.

Chapter 13

Optimizers

The most essential part of **MOSEK** are the optimizers:

- *primal simplex* (linear problems),
- *dual simplex* (linear problems),
- *interior-point* (linear, quadratic and conic problems),
- *mixed-integer* (problems with integer variables).

The structure of a successful optimization process is roughly:

- **Presolve**
  1. *Elimination*: Reduce the size of the problem.
  2. *Dualizer*: Choose whether to solve the primal or the dual form of the problem.
  3. *Scaling*: Scale the problem for better numerical stability.
- **Optimization**
  1. *Optimize*: Solve the problem using selected method.
  2. *Terminate*: Stop the optimization when specific termination criteria have been met.
  3. *Report*: Return the solution or an infeasibility certificate.

The preprocessing stage is transparent to the user, but useful to know about for tuning purposes. The purpose of the preprocessing steps is to make the actual optimization more efficient and robust. We discuss the details of the above steps in the following sections.

13.1 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, we suggest to first turn the eliminator off by setting the parameter *presolveEliminatorMaxNumTries* to 0. If that does not help, then trying to turn entire presolve off may help.

Since all computations are done in finite precision, the presolve employs some tolerances when concluding a variable is fixed or a 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 practice to build models without linear dependencies, but that is not always easy for the user to control. 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 more efficient to solve the primal or dual problem. The form (primal or dual) is displayed in the **MOSEK** log and available as an information item from the solver. 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 and conic (but not semidefinite) problems may be automatically 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 data. 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.

13.2 Linear Optimization

13.2.1 Optimizer Selection

Two different types of optimizers are available for linear problems: The default is an interior-point method, and the alternative is the simplex method (primal or dual). The optimizer can be selected using the parameter *optimizer*.

The Interior-point or the Simplex Optimizer?

Given a linear optimization problem, which optimizer is the best: the 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. On the other hand the simplex method can take advantage of an initial solution, but is less predictable from cold-start. The interior-point optimizer is used by default.

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, make it faster on average than the primal version. Still, it depends much on the problem structure and size. Setting the *optimizer* parameter to *"freeSimplex"* instructs **MOSEK** to choose one of the simplex variants automatically.

To summarize, if you want to know which optimizer is faster for a given problem type, it is best to try all the options.

13.2.2 The Interior-point Optimizer

The purpose of this section is to provide information about the algorithm employed in the **MOSEK** interior-point optimizer for linear problems and about its termination criteria.

The homogeneous primal-dual problem

In order to keep the discussion simple it is assumed that **MOSEK** solves linear optimization problems of standard form

$$\begin{array}{lccl} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \geq 0. & & \end{array} \tag{13.1}$$

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 (13.1) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason why **MOSEK** solves the so-called homogeneous model

$$\begin{array}{rcl} Ax - b\tau & = & 0, \\ A^T y + s - c\tau & = & 0, \\ -c^T x + b^T y - \kappa & = & 0, \\ x, s, \tau, \kappa & \geq & 0, \end{array} \tag{13.2}$$

where y and s correspond to the dual variables in (13.1), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.2) 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 (13.2) 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 (see Sec. 12.1 for the mathematical background on duality and optimality).

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

$$c^T x^* < 0 \tag{13.3}$$

or

$$b^T y^* > 0 \tag{13.4}$$

is satisfied. If (13.3) is satisfied then x^* is a certificate of dual infeasibility, whereas if (13.4) is satisfied then y^* is a certificate of primal 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 the k-th iteration of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to homogeneous model is generated, where

$$x^k, s^k, \tau^k, \kappa^k > 0.$$

Optimal case

Whenever the trial solution satisfies the criterion

$$\begin{aligned}
\left\| A\frac{x^k}{\tau^k} - b \right\|_\infty &\leq \epsilon_p (1 + \|b\|_\infty), \\
\left\| A^T \frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty &\leq \epsilon_d (1 + \|c\|_\infty), \text{ and} \\
\min\left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left|\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}\right| \right) &\leq \epsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right),
\end{aligned} \tag{13.5}$$

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 (13.5) 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.

Dual infeasibility certificate

On the other hand, if the trial solution satisfies

$$-\epsilon_i c^T x^k > \frac{\|c\|_\infty}{\max(1, \|b\|_\infty)} \left\| A x^k \right\|_\infty$$

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 $\left\| A x^k \right\|_\infty = 0$; then x^k is an exact certificate of dual infeasibility. Next assume that this is not the case, i.e.

$$\left\| A x^k \right\|_\infty > 0,$$

and define

$$\bar{x} := \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|A x^k\|_\infty \|c\|_\infty} x^k.$$

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.

Primal infeasibility certificate

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.

Adjusting optimality criteria

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see table for details.

Table 13.1: 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 (13.5) reveals that the quality of the solution depends 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, ε_g and ε_i, have to be relaxed together to achieve an effect.

If the optimizer terminates without locating a solution that satisfies the termination criteria, for example because of a stall or other numerical issues, then it will check if the solution found up to that point satisfies the same criteria with all tolerances multiplied by the value of *intpntCoTolNearRel*. If this is the case, the solution is still declared as optimal.

The basis identification discussed in Sec. 13.2.2 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 relaxations 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:

$$\begin{array}{lrcl} \text{minimize} & x + y & & \\ \text{subject to} & x + y & = & 1, \\ & x, y \geq 0. & & \end{array}$$

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-up* phase be short. However, for some cases of 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 (primal/dual) can be tuned with the parameter *biCleanOptimizer*, and the maximum number of iterations can be set with *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:

```
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 the 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 Sec. 13.2.2 the columns of the iteration log have the following meaning:

- `ITE`: Iteration index k.
- `PFEAS`: $\left\|Ax^k - b\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `DFEAS`: $\left\|A^Ty^k + s^k - c\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `GFEAS`: $|-c^Tx^k + b^Ty^k - \kappa^k|$. The numbers in this column should converge monotonically towards 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 to zero.
- TIME: Time spent since the optimization started.

13.2.3 The Simplex 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 Sec. 13.2.1 for a discussion. **MOSEK** provides both a primal and a dual variant of the simplex optimizer.

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 Sec. 12.1 for a definition of the primal and dual problem. Due to the fact that computations are performed in finite precision **MOSEK** allows violations of primal and dual feasibility within certain tolerances. The user can control the allowed primal and dual tolerances with the parameters *basisTolX* and *basisTolS*.

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. The same parameter can also be used to force one of the variants.

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.

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** treats a "numerically 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 a way to escape long sequences where the optimizer tries to recover from an unstable situation.

Examples of set-backs are: 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 it 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: 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 number of 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.

The Simplex Log

Below is a typical log output from the simplex optimizer:

```
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 667
Optimizer  - Scalar variables       : 1424            conic                  : 0
Optimizer  - hotstart               : no
ITER      DEGITER(%)  PFEAS       DFEAS       POBJ                  DOBJ                  TIME␣
↪     TOTTIME
0         0.00        1.43e+05    NA          6.5584140832e+03      NA                    0.00␣
↪     0.02
1000      1.10        0.00e+00    NA          1.4588289726e+04      NA                    0.13␣
↪     0.14
2000      0.75        0.00e+00    NA          7.3705564855e+03      NA                    0.21␣
↪     0.22
3000      0.67        0.00e+00    NA          6.0509727712e+03      NA                    0.29␣
↪     0.31
4000      0.52        0.00e+00    NA          5.5771203906e+03      NA                    0.38␣
↪     0.39
4533      0.49        0.00e+00    NA          5.5018458883e+03      NA                    0.42␣
↪     0.44
```

The first lines summarize the problem the optimizer is solving. This is followed by the iteration log, with the following meaning:

- `ITER`: Number of iterations.
- `DEGITER(%)`: Ratio of degenerate iterations.
- `PFEAS`: Primal feasibility measure reported by the simplex optimizer. The numbers should be 0 if the problem is primal feasible (when the primal variant is used).
- `DFEAS`: Dual feasibility measure reported by the simplex optimizer. The number should be 0 if the problem is dual feasible (when the dual variant is used).
- `POBJ`: An estimate for the primal objective value (when the primal variant is used).
- `DOBJ`: An estimate for the dual objective value (when the dual variant is used).
- `TIME`: Time spent since this instance of the simplex optimizer was invoked (in seconds).
- `TOTTIME`: Time spent since optimization started (in seconds).

13.3 Conic Optimization - Interior-point optimizer

For conic optimization problems only an interior-point type optimizer is available.

13.3.1 The homogeneous primal-dual problem

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]. In order to keep our discussion simple we will assume that **MOSEK** solves a conic optimization problem of the form:

$$\begin{array}{lccc} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \in \mathcal{K} & & \end{array} \tag{13.6}$$

where K is a convex cone. The corresponding dual problem is

$$\begin{array}{lccc} \text{maximize} & b^T y & & \\ \text{subject to} & A^T y + s & = & c, \\ & s \in \mathcal{K}^* & & \end{array} \tag{13.7}$$

where $\mathcal{K}^*$ is the dual cone of $\mathcal{K}$. See Sec. 12.2 for definitions.

Since it is not known beforehand whether problem (13.6) 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

$$\begin{array}{rcl} Ax - b\tau & = & 0, \\ A^T y + s - c\tau & = & 0, \\ -c^T x + b^T y - \kappa & = & 0, \\ x & \in & \mathcal{K}, \\ s & \in & \mathcal{K}^*, \\ \tau, \kappa & \geq & 0, \end{array} \tag{13.8}$$

where y and s correspond to the dual variables in (13.6), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.8) always has a 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 (13.8) satisfies

$$(x^*)^T s^* + \tau^* \kappa^* = 0$$

i.e. complementarity. Observe that $x^* \in \mathcal{K}$ and $s^* \in \mathcal{K}^*$ implies

$$(x^*)^T s^* \geq 0$$

and therefore

$$\tau^* \kappa^* = 0.$$

since $\tau^*, \kappa^* \geq 0$. Hence, at least one of τ^* and κ^* is zero.

First, assume that $\tau^* > 0$ and hence $\kappa^* = 0$. It follows that

$$\begin{array}{rcl} A \frac{x^*}{\tau^*} & = & b, \\ A^T \frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} & = & c, \\ -c^T \frac{x^*}{\tau^*} + b^T \frac{y^*}{\tau^*} & = & 0, \\ x^*/\tau^* & \in & \mathcal{K}, \\ s^*/\tau^* & \in & \mathcal{K}^*. \end{array}$$

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

$$\begin{array}{rcl} Ax^* & = & 0, \\ A^T y^* + s^* & = & 0, \\ -c^T x^* + b^T y^* & = & \kappa^*, \\ x^* & \in & \mathcal{K}, \\ s^* & \in & \mathcal{K}^*. \end{array}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.9}$$

or

$$b^T y^* > 0 \tag{13.10}$$

holds. If (13.9) is satisfied, then x^* is a certificate of dual infeasibility, whereas if (13.10) holds then y^* is a certificate of primal 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]*.

13.3.2 Interior-point Termination Criterion

Since computations are performed in finite precision, and for efficiency reasons, it is not possible to solve the homogeneous model exactly in general. 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 the homogeneous model is generated, where

$$x^k \in \mathcal{K}, s^k \in \mathcal{K}^*, \tau^k, \kappa^k > 0.$$

Therefore, it is possible to compute the values:

$$\begin{array}{rcl} \rho_p^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A \frac{x^k}{\tau^k} - b \right\|_\infty \leq \rho \varepsilon_p (1 + \|b\|_\infty) \right\}, \\ \rho_d^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A^T \frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty \leq \rho \varepsilon_d (1 + \|c\|_\infty) \right\}, \\ \rho_g^k & = & \arg\min_\rho \left\{ \rho \mid \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left| \frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k} \right| \right) \leq \rho \varepsilon_g \max \left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k} \right) \right\}, \\ \rho_{pi}^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A^T y^k + s^k \right\|_\infty \leq \rho \varepsilon_i b^T y^k, \, b^T y^k > 0 \right\} \text{ and} \\ \rho_{di}^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A x^k \right\|_\infty \leq -\rho \varepsilon_i c^T x^k, \, c^T x^k < 0 \right\}. \end{array}$$

Note $\varepsilon_p, \varepsilon_d, \varepsilon_g$ and ε_i are nonnegative user specified tolerances.

Optimal Case

Observe ρ_p^k measures how far x^k/τ^k is from being a good approximate primal feasible solution. Indeed if $\rho_p^k \leq 1$, then

$$\left\| A \frac{x^k}{\tau^k} - b \right\|_\infty \leq \varepsilon_p (1 + \|b\|_\infty). \tag{13.11}$$

This shows the violations in the primal equality constraints for the solution x^k/τ^k is small compared to the size of b given ε_p is small.

Similarly, if $\rho_d^k \leq 1$, then $(y^k, s^k)/\tau^k$ is an approximate dual feasible solution. If in addition $\rho_g \leq 1$, then the solution $(x^k, y^k, s^k)/\tau^k$ is approximate optimal because the associated primal and dual objective values are almost identical.

In other words if $\max(\rho_p^k, \rho_d^k, \rho_g^k) \leq 1$, then

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is an approximate optimal solution.

Dual Infeasibility Certificate

Next assume that $\rho_{di}^k \leq 1$ and hence

$$\left\|Ax^k\right\|_\infty \leq -\varepsilon_i c^T x^k \text{ and } -c^T x^k > 0$$

holds. Now in this case 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. Let

$$\bar{x} := \frac{x^k}{-c^T x^k}$$

and it is easy to verify that

$$\|A\bar{x}\|_\infty \leq \varepsilon_i \text{ and } c^T \bar{x} = -1$$

which shows $\bar{x}$ is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

Primal Infeasiblity Certificate

Next assume that $\rho_{pi}^k \leq 1$ and hence

$$\left\|A^T y^k + s^k\right\|_\infty \leq \varepsilon_i b^T y^k \text{ and } b^T y^k > 0$$

holds. Now in this case the problem is declared primal infeasible and (y^k, s^k) is reported as a certificate of primal infeasibility. The motivation for this stopping criterion is as follows. Let

$$\bar{y} := \frac{y^k}{b^T y^k} \text{ and } \bar{s} := \frac{s^k}{b^T y^k}$$

and it is easy to verify that

$$\left\|A^T \bar{y} + \bar{s}\right\|_\infty \leq \varepsilon_i \text{ and } b^T \bar{y} = 1$$

which shows (y^k, s^k) is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

13.3.3 Adjusting optimality criteria

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see table for details.

Table 13.2: Parameters employed in termination criterion

ToleranceParameter	name
ε_p	*intpntCoTolPfeas*
ε_d	*intpntCoTolDfeas*
ε_g	*intpntCoTolRelGap*
ε_i	*intpntCoTolInfeas*

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 (13.11) reveals that the quality of the solution depends 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, ε_g and ε_i, have to be relaxed together to achieve an effect.

If the optimizer terminates without locating a solution that satisfies the termination criteria, for example because of a stall or other numerical issues, then it will check if the solution found up to that point satisfies the same criteria with all tolerances multiplied by the value of *intpntCoTolNearRel*. If this is the case, the solution is still declared as optimal.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

13.3.4 The Interior-point Log

Below is a typical log output from the interior-point optimizer:

```
Optimizer  - threads                : 20
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 1
Optimizer  - Cones                  : 2
Optimizer  - Scalar variables       : 6                 conic                  : 6
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 : 1                 after factor           : 1
Factor     - dense dim.             : 0                 flops                  : 1.70e+01
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU       TIME
0   1.0e+00  2.9e-01  3.4e+00  0.00e+00   2.414213562e+00   0.000000000e+00   1.0e+00  0.01
1   2.7e-01  7.9e-02  2.2e+00  8.83e-01   6.969257574e-01   -9.685901771e-03  2.7e-01  0.01
2   6.5e-02  1.9e-02  1.2e+00  1.16e+00   7.606090061e-01   6.046141322e-01   6.5e-02  0.01
3   1.7e-03  5.0e-04  2.2e-01  1.12e+00   7.084385672e-01   7.045122560e-01   1.7e-03  0.01
4   1.4e-08  4.2e-09  4.9e-08  1.00e+00   7.071067941e-01   7.071067599e-01   1.4e-08  0.01
```

The first line displays the number of threads used by the optimizer and the 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 Sec. 13.3.1 the columns of the iteration log have the following meaning:

- `ITE`: Iteration index k.
- `PFEAS`: $\left\|Ax^k - b\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards 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 towards 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 towards 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 to zero.
- `TIME`: Time spent since the optimization started (in seconds).

13.4 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. Readers unfamiliar with integer optimization are recommended to consult some relevant literature, e.g. the book *[Wol98]* by Wolsey.

13.4.1 The Mixed-integer Optimizer Overview

MOSEK can solve mixed-integer linear and conic problems, except for mixed-integer semidefinite problems.

By default the mixed-integer optimizer is run-to-run deterministic. This means that if a problem is solved twice on the same computer with identical parameter settings and no time limit then the obtained solutions will be identical. 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:** See Sec. 13.1.
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.

13.4.2 Relaxations and bounds

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 (solving mixed-integer problems is NP-hard). For instance, a problem with n binary variables, may require time 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.

Consider for example a mixed-integer optimization problem

$$\begin{array}{rccccl} z^* & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \\ & & & x_j \in \mathbb{Z}, & & \forall j \in \mathcal{J}. \end{array} \tag{13.12}$$

It has the continuous relaxation

$$\begin{array}{rccccl} \underline{z} & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \end{array} \tag{13.13}$$

obtained simply by ignoring the integrality restrictions. The relaxation is a continuous problem, and therefore much faster to solve to optimality with a linear (or, in the general case, conic) optimizer. We call the optimal value $\underline{z}$ the *objective bound*. The objective bound $\underline{z}$ normally increases during the solution search process when the continuous relaxation is gradually refined.

Moreover, if $\hat{x}$ is any feasible solution to (13.12) and

$$\bar{z} := c^T \hat{x}$$

then

$$\underline{z} \leq z^* \leq \bar{z}.$$

These two inequalities allow us to estimate the quality of the integer solution: it 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.

13.4.3 Outer approximation for mixed-integer conic problems

The relaxations of mixed integer conic problems can be solved either as a nonlinear problem with the interior point algorithm (default) or with a linear outer approximation algorithm. The type of relaxation used can be set with *mioConicOuterApproximation*. The best value for this option is highly problem dependent.

13.4.4 Randomization

A number of internal algorithms of the mixed-integer solver are dependend on random tie-breaking. The random tie-breaking can have a significant impact on the path taken by the algorithm and the optimal solution returned. The random seed can be set with the parameter *mioSeed*.

13.4.5 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. The issue of terminating the mixed-integer optimizer is rather delicate and the user has numerous possibilities of influencing it with various parameters. 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

$$\bar{z} - \underline{z} \leq \max(\delta_2, \delta_3 \max(\delta_4, |\bar{z}|))$$

is satisfied. If this is the case, the integer optimizer terminates and reports the integer feasible solution as an optimal solution.

All the δ tolerances discussed above can be adjusted using suitable parameters — see Table 13.3.

Table 13.3: Tolerances for the mixed-integer optimizer.

Tolerance	Parameter name
δ_1	*mioTolAbsRelaxInt*
δ_2	*mioTolAbsGap*
δ_3	*mioTolRelGap*
δ_4	*mioRelGapConst*

In Table 13.4 some other common parameters affecting the integer optimizer termination criterion are shown.

Table 13.4: Other parameters affecting the integer optimizer termination criterion.

Parameter name	Explanation
mioMaxNumBranches	Maximum number of branches allowed.
mioMaxNumRelaxs	Maximum number of relaxations allowed.
mioMaxNumSolutions	Maximum number of feasible integer solutions allowed.

13.4.6 Speeding 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 Sec. 13.4.5 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. See Sec. 7.6.2.
- 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]*.

13.4.7 Understanding Solution Quality

To determine the quality of the solution one should check the following:

- The problem status and solution status returned by **MOSEK**, as well as constraint violations in case of suboptimal solutions.
- The *optimality gap* defined as

$$\epsilon = |(\text{objective value of feasible solution}) - (\text{objective bound})| = |\bar{z} - \underline{z}|.$$

  which measures how much the located solution can deviate from the optimal solution to the problem. The optimality gap can be retrieved through the information item `"mioObjAbsGap"`. Often it is more meaningful to look at the relative optimality gap normalized against the magnitude of the solution.

$$\epsilon_{\text{rel}} = \frac{|\bar{z} - \underline{z}|}{\max(\delta_4, |\bar{z}|)}.$$

  The relative optimality gap is available in the information item `"mioObjRelGap"`.

13.4.8 The Mixed-integer Log

Below is a typical log output from the mixed-integer optimizer:

```
Presolved problem: 6573 variables, 35728 constraints, 101258 non-zeros
Presolved problem: 0 general integer, 4294 binary, 2279 continuous
Clique table size: 1636
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(%)  TIME
0        1        0        0        NA                   1.8218819866e+07     NA          1.6
0        1        0        0        1.8331557950e+07     1.8218819866e+07     0.61        3.5
0        1        0        0        1.8300507546e+07     1.8218819866e+07     0.45        4.3
Cut generation started.
0        2        0        0        1.8300507546e+07     1.8218819866e+07     0.45        5.3
Cut generation terminated. Time = 1.43
0        3        0        0        1.8286893047e+07     1.8231580587e+07     0.30        7.5
15       18       1        0        1.8286893047e+07     1.8231580587e+07     0.30        10.5
31       34       1        0        1.8286893047e+07     1.8231580587e+07     0.30        11.1
51       54       1        0        1.8286893047e+07     1.8231580587e+07     0.30        11.6
91       94       1        0        1.8286893047e+07     1.8231580587e+07     0.30        12.4
171      174      1        0        1.8286893047e+07     1.8231580587e+07     0.30        14.3
331      334      1        0        1.8286893047e+07     1.8231580587e+07     0.30        17.9

[ ... ]

Objective of best integer solution : 1.825846762609e+07
Best objective bound               : 1.823311032986e+07
Construct solution objective       : Not employed
Construct solution # roundings     : 0
User objective cut value           : 0
Number of cuts generated           : 117
  Number of Gomory cuts            : 108
  Number of CMIR cuts              : 9
```

(continues on next page)

(continued from previous page)

```
Number of branches               : 4425
Number of relaxations solved     : 4410
Number of interior point iterations: 25
Number of simplex iterations     : 221131
```

The first lines contain a summary of the problem as seen by the optimizer. This is followed by the iteration log. The columns have the following meaning:

- `BRANCHES`: Number of branches generated.
- `RELAXS`: Number of relaxations solved.
- `ACT_NDS`: Number of active branch bound nodes.
- `DEPTH`: Depth of the recently solved node.
- `BEST_INT_OBJ`: The best integer objective value, $\bar{z}$.
- `BEST_RELAX_OBJ`: The best objective bound, $\underline{z}$.
- `REL_GAP(%)`: Relative optimality gap, $100\% \cdot \epsilon_{\mathrm{rel}}$
- `TIME`: Time (in seconds) from the start of optimization.

Following that a summary of the optimization process is printed.

Chapter 14

Fusion API Reference

- *General API conventions.*
- **mosek.fusion classes**
  - Quick links: *Model*, *Expr*, *Variable*, *Var*, *Domain*, *Matrix*
  - *Full list*
- **Optimizer parameters:**
  - *Double*, *Integer*, *String*
  - *Full list*
  - *Browse by topic*
- **Optimizer information items:**
  - *Double*, *Integer*, *Long*
- *Enumerations*
- *Constants*
- *Exceptions*
- *Linear algebra utilities*

14.1 *Fusion* API conventions

14.1.1 General conventions

All the classes of the *Fusion* interface are contained in the package **mosek.fusion**. The user should not directly instantiate objects of any class other than creating a *Model*.

```
M = Model()
M = Model('modelName')

with Model() as M:
```

The model is the main access point to an optimization problem and its solution. All other objects should be created through the model (*Model.variable*, *Model.constraint*, etc.) or using static factory methods (*Matrix.sparse* etc.).

Whenever a method expects a numeric or integer array, this can be either a Python list or a numpy array of appropriate shape.

14.2 Class list

Common

- *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.
- *Set*: Handles shapes.
- *Var*: Provides basic operations on variable objects.
- *Variable*: Abstract base class for Variable objects.

Infrequent

- *BaseExpression*: Base class for most expressions
- *BaseVariable*: Abstract base class for Variable objects with default implementations.
- *ConeDomain*: A domain representing a vector cone.
- *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.
- *NDSparseArray*: Representation of a sparse n-dimensional array.
- *PSDConstraint*: Represents a semidefinite conic constraint.
- *PSDDomain*: Represent the domain od PSD matrices.
- *PSDVariable*: This class represents a positive semidefinite variable.
- *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.

- *SymmetricLinearDomain*: Represent a linear domain with symmetry.
- *SymmetricRangeDomain*: Represent a ranged domain with symmetry.
- *SymmetricVariable*: An object representing a symmetric variable.
- *WorkStack*: Stack object used to store expression evaluations. For internal use.

14.2.1 Class BaseExpression

`mosek.fusion.BaseExpression`

Base class for most expressions

Members *BaseExpression.eval* – Evaluate the expression and push the result onto the work stack.
BaseExpression.getDim – Return the d'th dimension in the expression.
BaseExpression.getND – Return the number of dimensions in the expression.
BaseExpression.getShape – Get the shape of the expression.
BaseExpression.getSize – Return the total number of elements in the expression (the product of the dimensions).
BaseExpression.index – Get a single element in the expression.
BaseExpression.pick – Pick a number of elements from the expression.
BaseExpression.slice – Get a slice of the expression.
BaseExpression.toString – Return a string representation of the expression object.

Implemented by *Expr*

`BaseExpression.eval`

```
eval(WorkStack rs, WorkStack ws, WorkStack xs)
eval() -> FlatExpr
```

Evaluate the expression and push the result onto the `rs` work stack.

Parameters
- `rs` (*WorkStack*) – The stack where the result of the evaluation is stored.
- `ws` (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- `xs` (*WorkStack*)

Return (*FlatExpr*)

`BaseExpression.getDim`

```
getDim(int d) -> int
```

Return the d'th dimension in the expression.

Parameters `d` (int)
Return (int)

`BaseExpression.getND`

```
getND() -> int
```

Return the number of dimensions in the expression.

Return (int)

`BaseExpression.getShape`

```
getShape() -> int[]
```

Get the shape of the expression.

Return (int[])

`BaseExpression.getSize`

```
getSize() -> int
```

Return the total number of elements in the expression (the product of the dimensions).

Return (int)

`BaseExpression.index`

```
index(int i) -> Expression
index(int[] indexes) -> Expression
```

Get a single element in the expression.

Parameters

- `i` (int) – Index of the element to pick.
- `indexes` (int[]) – Multi-dimensional index of the element to pick.

Return (*Expression*)

`BaseExpression.pick`

```
pick(int[] indexes) -> Expression
pick(int[][] indexrows) -> Expression
```

Picks a number of elements from the expression and returns them as a one-dimensional expression.

Parameters

- `indexes` (int[]) – Indexes of the elements to pick
- `indexrows` (int[][]) – Indexes of the elements to pick. Each row defines a separate multi-dimensional index.

Return (*Expression*)

`BaseExpression.slice`

```
slice(int first, int last) -> Expression
slice(int[] firsta, int[] lasta) -> Expression
```

Get a slice of the expression.

Parameters

- `first` (int) – Index of the first element in the slice.
- `last` (int) – Index of the last element in the slice plus one.
- `firsta` (int[]) – Multi-dimensional index of the first element in the slice.
- `lasta` (int[]) – Multi-dimensional index of the element after the end of the slice.

Return (*Expression*)

`BaseExpression.toString`

```
toString() -> str
```

Return a string representation of the expression object.

Return (str)

14.2.2 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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this 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 *SliceVariable*, *ModelVariable*

BaseVariable.antidiag

```
antidiag() -> Variable
antidiag(int index) -> Variable
```

Return the antidiagonal of a square variable matrix.

Parameters index (int) – Index of the anti-diagonal

Return (*Variable*)

BaseVariable.asExpr

```
asExpr() -> Expression
```

Create an *Expression* object corresponding to $I \cdot V$, where I is the identity matrix and V is this variable.

Return (*Expression*)

`BaseVariable.diag`

```
diag() -> Variable
diag(int index) -> Variable
```

Return the diagonal of a square variable matrix.

Parameters `index` (`int`) – Index of the anti-diagonal
Return (*Variable*)

`BaseVariable.dual`

```
dual() -> float[]
```

Get the dual solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (`float`[])

`BaseVariable.eval`

```
eval(WorkStack rs, WorkStack ws, WorkStack xs)
eval() -> FlatExpr
```

Evaluate the expression and push the result onto the `rs` work stack.

Parameters

- `rs` (*WorkStack*) – The stack where the result of the evaluation is stored.
- `ws` (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- `xs` (*WorkStack*)

Return (*FlatExpr*)

`BaseVariable.getDim`

```
getDim(int d) -> int
```

Return the d'th dimension in the expression.

Parameters `d` (`int`)
Return (`int`)

`BaseVariable.getModel`

```
getModel() -> Model
```

Get the *Model* object that the variable belongs to.

Return (*Model*)

`BaseVariable.getND`

```
getND() -> int
```

Get the number of dimensions in the variable shape.

Return (`int`)

`BaseVariable.getShape`

```
getShape() -> int[]
```

Get the variable shape.

Return (int[])

`BaseVariable.getSize`

```
getSize() -> int
```

Get the total number of elements in the variable.

Return (int)

`BaseVariable.index`

```
index(int index) -> Variable
index(int[] index) -> Variable
index(int i0, int i1) -> Variable
index(int i0, int i1, int i2) -> Variable
```

Return a variable slice of size 1 corresponding to a single element in the variable object..

Parameters

- `index` (int)
- `index` (int[])
- `i0` (int) – Index in the first dimension of the element requested.
- `i1` (int) – Index in the second dimension of the element requested.
- `i2` (int) – Index in the second dimension of the element requested.

Return (*Variable*)

`BaseVariable.level`

```
level() -> float[]
```

Get the primal solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (float[])

`BaseVariable.makeContinuous`

```
makeContinuous()
```

Drop integrality constraints on the variable, if any.

`BaseVariable.makeInteger`

```
makeInteger()
```

Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.

`BaseVariable.pick`

```
pick(int[] idxs) -> Variable
pick(int[][] midxs) -> Variable
pick(int[] i0, int[] i1) -> Variable
pick(int[] i0, int[] i1, int[] i2) -> Variable
```

Create a slice variable by picking a list of indexes from this variable.

Parameters
- `idxs` (`int`[]) – Indexes of the elements requested.
- `midxs` (`int`[][]) – Matrix of indexes of the elements requested.
- `i0` (`int`[])
- `i1` (`int`[]) – Index along the first dimension.
- `i2` (`int`[]) – Index along the second dimension.

Return (*Variable*)

`BaseVariable.reshape`

```
reshape(int[] shape) -> Variable
reshape(int dim0) -> Variable
reshape(int dim0, int dim1) -> Variable
reshape(int dim0, int dim1, int dim2) -> Variable
```

Reshape the variable. The new shape must have the same total size as the current.

Parameters
- `shape` (`int`[]) – The new shape.
- `dim0` (`int`) – First dimension of new shape
- `dim1` (`int`) – Second dimension of new shape
- `dim2` (`int`) – Third dimension of new shape

Return (*Variable*)

`BaseVariable.setLevel`

```
setLevel(float[] v)
```

Set values for an initial solution for this variable. Note that these values are buffered until the solver is called; they are not available through the `level()` methods.

Parameters `v` (`float`[]) – An array of values to be assigned to the variable.

`BaseVariable.slice`

```
slice(int first, int last) -> Variable
slice(int[] first, int[] last) -> Variable
```

Create a slice variable by picking a range of indexes for each variable dimension

Parameters
- `first` (`int`) – The index of the first element(s) of the slice.
- `first` (`int`[]) – The index of the first element(s) of the slice.
- `last` (`int`) – The index after the last element of the slice.
- `last` (`int`[])

Return (*Variable*)

`BaseVariable.toString`

```
toString() -> str
```

Create a string representation of the variable.

Return (str)

BaseVariable.transpose

```
transpose() -> Variable
```

Note that this requires a one- or two-dimensional variable.

Return (*Variable*)

14.2.3 Class ConeDomain

mosek.fusion.ConeDomain
A domain representing a vector cone.

Members *ConeDomain.axis* – Set the dimension along which the cones are created.
ConeDomain.axisIsSet – Returns true if the cone axis was set
ConeDomain.getAxis – Get the dimension along which the cones are created.
ConeDomain.integral – Creates a domain of integral variables.

ConeDomain.axis

```
axis(int a) -> ConeDomain
```

Set the dimension along which the cones are created.

Parameters a (int)
Return (*ConeDomain*)

ConeDomain.axisIsSet

```
axisIsSet() -> bool
```

Returns true if the cone axis was set

Return (bool)

ConeDomain.getAxis

```
getAxis() -> int
```

Get the dimension along which the cones are created.

Return (int)

ConeDomain.integral

```
integral() -> ConeDomain
```

Modify a given domain restricting its elements to be integral.

Return (*ConeDomain*)

14.2.4 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.

Implements *ModelConstraint*

Members *ConicConstraint.toString* – Create a human readable string representation of the constraint.

Constraint.dual – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.slice – Create a slice constraint.

Constraint.update – Update part of a constraint.

ConicConstraint.toString

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

14.2.5 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.

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.

BaseVariable.getSize – Get the total number of elements in the variable.

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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this variable
BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension
BaseVariable.transpose – Transpose a vector or matrix variable
ConicVariable.toString – Create a string representation of the variable.

`ConicVariable.toString`

```
toString() -> str
```

Create a string representation of the variable.

Return (str)

14.2.6 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.dual* – Get the dual solution values of the constraint.
Constraint.getModel – Return the model that the constraint belongs to.
Constraint.getND – Return the number of dimensions in the constraint shape.
Constraint.getShape – Return the constraint's shape.
Constraint.getSize – Return the total number of elements in the constraint.
Constraint.index – Get a single element from a constraint.
Constraint.level – Get the primal solution values of the constraint.
Constraint.slice – Create a slice constraint.
Constraint.toString – Create a human readable string representation of the constraint.
Constraint.update – Update part of a constraint.

Static members *Constraint.hstack* – Stack a number of constraints horizontally.
Constraint.stack – Stack a number of constraints.
Constraint.vstack – Stack a number of constraints vertically.

Implemented by *ModelConstraint* , *SliceConstraint*

`Constraint.dual`

```
dual() -> float[]
```

Get the dual solution values of the constraint or its slice. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (float[])

`Constraint.getModel`

```
getModel() -> Model
```

Return the model that the constraint belongs to.

Return (*Model*)

`Constraint.getND`

```
getND() -> int
```

Return the number of dimensions in the constraint shape.

Return (int)

`Constraint.getShape`

```
getShape() -> int[]
```

Return the constraint's shape.

Return (int[])

`Constraint.getSize`

```
getSize() -> int
```

Return the total number of elements in the constraint.

Return (int)

`Constraint.hstack`

```
Constraint.hstack(Constraint v1, Constraint v2) -> Constraint
Constraint.hstack(Constraint v1, Constraint v2, Constraint v3) -> Constraint
Constraint.hstack(Constraint[] clist) -> Constraint
```

Stack a number of constraints horizontally.

Parameters

- `v1` (*Constraint*) – The first constraint in the stack.
- `v2` (*Constraint*) – The second constraint in the stack.
- `v3` (*Constraint*) – The third constraint in the stack.
- `clist` (*Constraint*[]) – The constraints in the stack.

Return (*Constraint*)

`Constraint.index`

```
index(int idx) -> Constraint
index(int[] idxa) -> Constraint
```

Get a single element from a one-dimensional constraint.

Parameters

- `idx` (int) – The index of the element.
- `idxa` (int[]) – A multi-dimensional index of the element.

Return (*Constraint*)

`Constraint.level`

```
level() -> float[]
```

Get the primal solution values of the constraint. This amounts to evaluating the Ax part of the constraint expression for the relevant solution value. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (float[])

`Constraint.slice`

```
slice(int first, int last) -> Constraint
slice(int[] firsta, int[] lasta) -> Constraint
```

Create a slice constraint.

Parameters

- `first` (int) – Index of the first element in the slice.
- `last` (int) – Index of the first element after the end of the slice.
- `firsta` (int[]) – The indexes of first elements in the slice along each dimension.
- `lasta` (int[]) – The indexes of first elements after the end of the slice along each dimension.

Return (*Constraint*)

`Constraint.stack`

```
Constraint.stack(Constraint v1, Constraint v2, int dim) -> Constraint
Constraint.stack(Constraint v1, Constraint v2, Constraint v3, int dim) -> Constraint
Constraint.stack(Constraint[] clist, int dim) -> Constraint
```

Stack a number of constraints.

Parameters

- `v1` (*Constraint*) – The first constraint in the stack.
- `v2` (*Constraint*) – The second constraint in the stack.
- `dim` (int) – The dimension in which to stack, 0 means vertically.
- `v3` (*Constraint*) – The third constraint in the stack.
- `clist` (*Constraint*[]) – The constraints in the stack.

Return (*Constraint*)

`Constraint.toString`

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

`Constraint.update`

```
update(Expression expr, Variable x)
update(Expression expr, Variable x, bool bfixupdate)
update(Expression expr)
update(float[] bfix)
```

Update entire or part of the left-hand side of a constraint (the expression). See Sec. 7.9 for a tutorial.

If only `expr` is given, replace the entire previous expression with `expr`. The shape of `expr` must match the shape of the constraint.

If only `bfix` is given, replace the constant term of the expression with `bfix`. The length of `bfix` must match the size of the expression.

If `x` is given, replace all columns in the constraint defined by `x` by terms defined by `expr`, possibly including constant terms. Currently it is only possible to update linear columns. Attempting to update columns of semidefinite variables will result in an error.

Parameters

- `expr` (*Expression*) – The expression to update with.
- `x` (*Variable*) – Variable object defining the columns to update.
- `bfixupdate` (`bool`) – Whether to include fixed terms as well.
- `bfix` (`float`[]) – The fixed term to update with.

`Constraint.vstack`

```
Constraint.vstack(Constraint v1, Constraint v2) -> Constraint
Constraint.vstack(Constraint v1, Constraint v2, Constraint v3) -> Constraint
Constraint.vstack(Constraint[] clist) -> Constraint
```

Stack a number of constraints vertically.

Parameters

- `v1` (*Constraint*) – The first constraint in the stack.
- `v2` (*Constraint*) – The second constraint in the stack.
- `v3` (*Constraint*) – The third constraint in the stack.
- `clist` (*Constraint*[]) – The constraints in the stack.

Return (*Constraint*)

14.2.7 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* and *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 specified by a lower bound in each dimension.

Domain.inDExpCone – Defines the dual exponential cone.

Domain.inDPowerCone – Defines the dual power cone.

Domain.inPExpCone – Defines the primal exponential cone.

Domain.inPPowerCone – Defines the primal power cone.

Domain.inPSDCone – Creates a domain of Positive Semidefinite matrices.

Domain.inQCone – Defines the domain of quadratic cones.

Domain.inRange – Creates a domain specified by a range in each dimension.

Domain.inRotatedQCone – Defines the domain of rotated 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 specified by an upper bound in each dimension.

Domain.sparse – Use a sparse representation.
Domain.symmetric – Impose symmetry on a given linear domain.
Domain.unbounded – Creates a domain in which variables are unbounded.

`Domain.axis`

```
Domain.axis(ConeDomain c, int a) -> ConeDomain
```

Set the dimension along which the cones are created. If this conic domain is used for a variable or expression of dimension d, then the conic constraint will be applicable to all vectors obtained by fixing the coordinates other than a-th and moving along the a-th coordinate. If $d = 2$ this can be used to define the conditions "every row of the matrix is in a cone" and "every column of a matrix is in a cone".

The default is the last dimension $a = d - 1$.

Parameters

- `c` (*ConeDomain*) – A conic domain.
- `a` (`int`) – The axis.

Return (*ConeDomain*)

`Domain.binary`

```
Domain.binary(int n) -> RangeDomain
Domain.binary(int m, int n) -> RangeDomain
Domain.binary(int[] dims) -> RangeDomain
Domain.binary() -> RangeDomain
```

Create a domain of binary variables. A binary domain can only be used when creating variables, but is not allowed in a constraint. Another way of restricting variables to be integers is the method *Variable.makeInteger*.

Parameters

- `n` (`int`) – Dimension size.
- `m` (`int`) – Dimension size.
- `dims` (`int`[]) – A list of dimension sizes.

Return (*RangeDomain*)

`Domain.equalsTo`

```
Domain.equalsTo(float b) -> LinearDomain
Domain.equalsTo(float b, int n) -> LinearDomain
Domain.equalsTo(float b, int m, int n) -> LinearDomain
Domain.equalsTo(float b, int[] dims) -> LinearDomain
Domain.equalsTo(float[] a1) -> LinearDomain
Domain.equalsTo(float[][] a2) -> LinearDomain
Domain.equalsTo(float[] a1, int[] dims) -> LinearDomain
Domain.equalsTo(Matrix mx) -> LinearDomain
```

Defines the domain consisting of a fixed point.

Parameters

- `b` (`float`) – A single value. This is scalable: it means that each element in the variable or constraint is fixed to b.
- `n` (`int`) – Dimension size.
- `m` (`int`) – Dimension size.
- `dims` (`int`[]) – A list of dimension sizes.
- `a1` (`float`[]) – A one-dimensional array of bounds. The shape must match the variable or constraint with which it is used.

- a2 (float[][]) – A two-dimensional array of bounds. The 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 (*LinearDomain*)

Domain.greaterThan

```
Domain.greaterThan(float b) -> LinearDomain
Domain.greaterThan(float b, int n) -> LinearDomain
Domain.greaterThan(float b, int m, int n) -> LinearDomain
Domain.greaterThan(float b, int[] dims) -> LinearDomain
Domain.greaterThan(float[] a1) -> LinearDomain
Domain.greaterThan(float[][] a2) -> LinearDomain
Domain.greaterThan(float[] a1, int[] dims) -> LinearDomain
Domain.greaterThan(Matrix mx) -> LinearDomain
```

Defines the domain specified by a lower bound in each dimension.

Parameters

- b (float) – A single value. This is scalable: it means that each element in the variable or constraint is greater than or equal to b.
- n (int) – Dimension size.
- m (int) – Dimension size.
- dims (int[]) – A list of dimension sizes.
- a1 (float[]) – A one-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- a2 (float[][]) – A two-dimensional array of bounds. The 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 (*LinearDomain*)

Domain.inDExpCone

```
Domain.inDExpCone() -> ConeDomain
Domain.inDExpCone(int m) -> ConeDomain
Domain.inDExpCone(int[] dims) -> ConeDomain
```

Defines the domain of dual exponential cones:

$$\left\{x \in \mathbb{R}^3 \ : \ x_1 \geq -x_3 e^{-1} e^{x_2/x_3},\ x_1 > 0,\ x_3 < 0\right\}$$

The conic domain scales as follows. If a variable or expression constrained to an exponential cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d = 2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- m (int) – The number of cones (default 1).
- dims (int[]) – Shape of the domain.

Return (*ConeDomain*)

Domain.inDPowerCone

```
Domain.inDPowerCone(float alpha) -> ConeDomain
Domain.inDPowerCone(float alpha, int m) -> ConeDomain
Domain.inDPowerCone(float alpha, int[] dims) -> ConeDomain
```

Defines the domain of dual power cones:

$$\left\{x \in \mathbb{R}^n \ : \ \left(\frac{x_1}{\alpha}\right)^\alpha \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{i=3}^n x_i^2}, \ x_1, x_2 \geq 0\right\}$$

The conic domain scales as follows. If a variable or expression constrained to a power cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d=2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- `alpha` (`float`) – The exponent of the power cone.
- `m` (`int`) – The number of cones (default 1).
- `dims` (`int[]`) – Shape of the domain.

Return (*ConeDomain*)

`Domain.inPExpCone`

```
Domain.inPExpCone() -> ConeDomain
Domain.inPExpCone(int m) -> ConeDomain
Domain.inPExpCone(int[] dims) -> ConeDomain
```

Defines the domain of primal exponential cones:

$$\left\{x \in \mathbb{R}^3 \ : \ x_1 \geq x_2 e^{x_3/x_2}, \ x_1, x_2 > 0\right\}$$

The conic domain scales as follows. If a variable or expression constrained to an exponential cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d=2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- `m` (`int`) – The number of cones (default 1).
- `dims` (`int[]`) – Shape of the domain.

Return (*ConeDomain*)

`Domain.inPPowerCone`

```
Domain.inPPowerCone(float alpha) -> ConeDomain
Domain.inPPowerCone(float alpha, int m) -> ConeDomain
Domain.inPPowerCone(float alpha, int[] dims) -> ConeDomain
```

Defines the domain of primal power cones:

$$\left\{x \in \mathbb{R}^n \ : \ x_1^\alpha x_2^{1-\alpha} \geq \sqrt{\sum_{i=3}^n x_i^2}, \ x_1, x_2 \geq 0\right\}$$

The conic domain scales as follows. If a variable or expression constrained to a power cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d=2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- alpha (float) – The exponent of the power cone.
- m (int) – The number of cones (default 1).
- dims (int[]) – Shape of the domain.

Return (*ConeDomain*)

`Domain.inPSDCone`

```
Domain.inPSDCone() -> PSDDomain
Domain.inPSDCone(int n) -> PSDDomain
Domain.inPSDCone(int n, int m) -> PSDDomain
```

When used to create a new variable in *Model.variable* it defines a domain of symmetric positive semidefinite matrices, that is

$$\mathcal{S}_+^n = \left\{X \in \mathbb{R}^{n\times n} \ : \ X = X^T, \ y^T X y \geq 0, \ \text{for all } y\right\}.$$

The shape of the result is $n \times n$. If m was given the domain is a product of m such cones, that is of shape $m \times n \times n$.

When used to impose a constraint in *Model.constraint* it defines a domain

$$\left\{X \in \mathbb{R}^{n\times n} \ : \ \frac{1}{2}(X + X^T) \in \mathcal{S}_+^n\right\}.$$

i.e. a positive semidefinite matrix without the symmetry assumption.

Parameters

- n (int) – Dimension of the PSD matrix.
- m (int) – Number of matrices (default 1).

Return (*PSDDomain*)

`Domain.inQCone`

```
Domain.inQCone() -> ConeDomain
Domain.inQCone(int n) -> ConeDomain
Domain.inQCone(int m, int n) -> ConeDomain
Domain.inQCone(int[] dims) -> ConeDomain
```

Defines the domain of quadratic cones:

$$\left\{x \in \mathbb{R}^n \ : \ x_1^2 \geq \sum_{i=2}^{n} x_i^2, \ x_1 \geq 0\right\}$$

The conic domain scales as follows. If a variable or expression constrained to a quadratic cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d = 2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- n (int) – The size of each cone; at least 2.
- m (int) – The number of cones (default 1).
- dims (int[]) – Shape of the domain.

Return (*ConeDomain*)

`Domain.inRange`

```
Domain.inRange(float lb, float ub) -> RangeDomain
Domain.inRange(float lb, float[] uba) -> RangeDomain
Domain.inRange(float[] lba, float ub) -> RangeDomain
Domain.inRange(float[] lba, float[] uba) -> RangeDomain
Domain.inRange(float lb, float ub, int[] dims) -> RangeDomain
Domain.inRange(float lb, float[] uba, int[] dims) -> RangeDomain
Domain.inRange(float[] lba, float ub, int[] dims) -> RangeDomain
Domain.inRange(float[] lba, float[] uba, int[] dims) -> RangeDomain
```

Creates a domain specified by a range in each dimension.

Parameters

- `lb` (`float`) – The lower bound as a common scalar value.
- `ub` (`float`) – The upper bound as a common scalar value.
- `uba` (`float`[]) – The upper bounds as an array.
- `lba` (`float`[]) – The lower bounds as an array.
- `dims` (`int`[]) – A list of dimension sizes.

Return (*RangeDomain*)

`Domain.inRotatedQCone`

```
Domain.inRotatedQCone() -> ConeDomain
Domain.inRotatedQCone(int n) -> ConeDomain
Domain.inRotatedQCone(int m, int n) -> ConeDomain
Domain.inRotatedQCone(int[] dims) -> ConeDomain
```

Defines the domain of rotated quadratic cones:

$$\left\{ x \in \mathbb{R}^n \ : \ 2x_1x_2 \geq \sum_{i=3}^{n} x_i^2, \ x_1, x_2 \geq 0 \right\}$$

The conic domain scales as follows. If a variable or expression constrained to a quadratic cone is not a single vector but a d-dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d-1$ coordinates and moving along the last coordinate. If $d = 2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- `n` (`int`) – The size of each cone; at least 3.
- `m` (`int`) – The number of cones (default 1).
- `dims` (`int`[]) – Shape of the domain.

Return (*ConeDomain*)

`Domain.integral`

```
Domain.integral(ConeDomain c) -> ConeDomain
Domain.integral(LinearDomain ld) -> LinearDomain
Domain.integral(RangeDomain rd) -> RangeDomain
```

Modify a given domain restricting its elements to be integral. An integral domain can only be used when creating variables, but is not allowed in a constraint. Another way of restricting variables to be integers is the method *Variable.makeInteger*.

Parameters

- `c` (*ConeDomain*) – A conic domain.
- `ld` (*LinearDomain*) – A linear domain.
- `rd` (*RangeDomain*) – A ranged domain.

Return

- (*ConeDomain*)
- (*LinearDomain*)
- (*RangeDomain*)

`Domain.isLinPSD`

```
Domain.isLinPSD() -> LinPSDDomain
Domain.isLinPSD(int n) -> LinPSDDomain
Domain.isLinPSD(int n, int m) -> LinPSDDomain
```

Creates an domain of vectors of length $\frac{1}{2}n(n+1)$ which are the flattenings of the lower-triangular part of a symmetric positive-semidefinite matrix X. The shape of the result is $\frac{1}{2}n(n+1)$. If m was given the domain is a product of m such domains, that is of shape $\frac{1}{2}n(n+1)m$.

Parameters

- `n` (`int`) – Dimension of the PSD matrix.
- `m` (`int`) – Number of matrices (default 1).

Return (*LinPSDDomain*)

`Domain.isTrilPSD`

```
Domain.isTrilPSD() -> PSDDomain
Domain.isTrilPSD(int n) -> PSDDomain
Domain.isTrilPSD(int n, int m) -> PSDDomain
```

Creates an object representing a cone 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 defines the symmetric matrix that is positive semidefinite. The shape of the result is $n \times n$. If m was given the domain is a product of m such cones, that is of shape $m \times n \times n$.

Parameters

- `n` (`int`) – Dimension of the PSD matrix.
- `m` (`int`) – Number of matrices (default 1).

Return (*PSDDomain*)

`Domain.lessThan`

```
Domain.lessThan(float b) -> LinearDomain
Domain.lessThan(float b, int n) -> LinearDomain
Domain.lessThan(float b, int m, int n) -> LinearDomain
Domain.lessThan(float b, int[] dims) -> LinearDomain
Domain.lessThan(float[] a1) -> LinearDomain
Domain.lessThan(float[][] a2) -> LinearDomain
Domain.lessThan(float[] a1, int[] dims) -> LinearDomain
Domain.lessThan(Matrix mx) -> LinearDomain
```

Defines the domain specified by an upper bound in each dimension.

Parameters

- `b` (`float`) – A single value. This is scalable: it means that each element in the variable or constraint is less than or equal to b.
- `n` (`int`) – Dimension size.
- `m` (`int`) – Dimension size.
- `dims` (`int[]`) – A list of dimension sizes.

- a1 (float[]) – A one-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- a2 (float[][]) – A two-dimensional array of bounds. The 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 (*LinearDomain*)

`Domain.sparse`

```
Domain.sparse(LinearDomain ld, int[] sparsity) -> LinearDomain
Domain.sparse(LinearDomain ld, int[][] sparsity) -> LinearDomain
Domain.sparse(RangeDomain rd, int[] sparsity) -> RangeDomain
Domain.sparse(RangeDomain rd, int[][] sparsity) -> RangeDomain
```

Given a linear domain, this method explicitly suggest to *Fusion* that a sparse representation is helpful.

Parameters

- ld (*LinearDomain*) – The linear sparse domain.
- sparsity (int[]) – Sparsity pattern.
- sparsity (int[][]) – Sparsity pattern.
- rd (*RangeDomain*) – The ranged sparse domain.

Return

- (*LinearDomain*)
- (*RangeDomain*)

`Domain.symmetric`

```
Domain.symmetric(LinearDomain ld) -> SymmetricLinearDomain
Domain.symmetric(RangeDomain rd) -> SymmetricRangeDomain
```

Given a linear domain D whose shape is that of square matrices, this method returns a domain consisting of symmetric matrices in D.

Parameters

- ld (*LinearDomain*) – The linear domain to be symmetrized.
- rd (*RangeDomain*) – The ranged domain to be symmetrized.

Return

- (*SymmetricLinearDomain*)
- (*SymmetricRangeDomain*)

`Domain.unbounded`

```
Domain.unbounded() -> LinearDomain
Domain.unbounded(int n) -> LinearDomain
Domain.unbounded(int m, int n) -> LinearDomain
Domain.unbounded(int[] dims) -> LinearDomain
```

Creates a domain in which variables are unbounded.

Parameters

- n (int) – Dimension size.
- m (int) – Dimension size.
- dims (int[]) – A list of dimension sizes.

Return (*LinearDomain*)

14.2.8 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 methods for constructing and manipulating various expressions.

Implements *BaseExpression*

Members *BaseExpression.getDim* – Return the d'th dimension in the expression.

BaseExpression.getSize – Return the total number of elements in the expression (the product of the dimensions).

BaseExpression.index – Get a single element in the expression.

BaseExpression.pick – Pick a number of elements from the expression.

BaseExpression.slice – Get a slice of the expression.

BaseExpression.toString – Return a string representation of the expression object.

Expr.eval – Evaluate the expression and push the result onto the work stack.

Expr.getND – Return the number of dimensions in the expression.

Expr.getShape – Return the shape of the expression

Expr.numNonzeros – Return the number of non-zero elements in the expression.

Expr.size – Return the expression size.

Static members *Expr.add* – Compute the sum of expressions.

Expr.condense – Flatten expression and remove all structural zeros.

Expr.constTerm – Create an expression consisting of a constant vector of values.

Expr.dot – Return a scalar expression object representing the dot-product of two items.

Expr.flatten – Reshape 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 matrices.

Expr.mulElm – Element-wise product of two items.

Expr.neg – Change the sign of an expression

Expr.ones – Create an expression consisting of ones.

Expr.outer – Return the outer-product of two vectors.

Expr.repeat – Repeat 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 – Compute the difference of two expressions.

Expr.sum – Sum the elements of an expression.

Expr.transpose – Transpose a two-dimensional expression.

Expr.vstack – Stack a list of expressions vertically (i.e. along the first dimension).

Expr.zeros – Create an expression consisting of zeros.

`Expr.add`

```
Expr.add(Expression e1, Expression e2) -> Expression
Expr.add(Expression e1, float[] a1) -> Expression
Expr.add(Expression e1, float[][] a2) -> Expression
Expr.add(float[] a1, Expression e2) -> Expression
Expr.add(float[][] a2, Expression e2) -> Expression
Expr.add(Expression e1, float c) -> Expression
Expr.add(float c, Expression e2) -> Expression
Expr.add(Expression e1, Matrix m) -> Expression
Expr.add(Matrix m, Expression e2) -> Expression
Expr.add(Expression e1, NDSparseArray n) -> Expression
```

(continues on next page)

(continued from previous page)

```
Expr.add(NDSparseArray n, Expression e2) -> Expression
Expr.add(Variable[] vs) -> Expression
Expr.add(Expression[] exps) -> Expression
```

Computes the sum of two or more expressions or variables. The following types of arguments are allowed:

A	B
Variable	Variable
Expression	Expression
double	
double[]	
double[,]	
Matrix	
NDSparseArray	

By symmetry both `add(A,B)` and `add(B,A)` are available.

The arguments must have the same shapes and the returned expression also has that shape. If one of the arguments is a single scalar, it is promoted to the shape that matches the shape of the other argument, i.e. the scalar is added to all entries of the other argument.

Parameters

- `e1` (*Expression*) – An expression.
- `e2` (*Expression*) – An expression.
- `a1` (`float`[]) – A one-dimensional array of constants.
- `a2` (`float`[][]) – A two-dimensional array of constants.
- `c` (`float`) – A constant.
- `m` (*Matrix*) – A Matrix object.
- `n` (*NDSparseArray*) – An NDSparseArray object.
- `vs` (*Variable*[]) – A list of variables. All variables in the array must have the same shape. The list must contain at least one element.
- `exps` (*Expression*[]) – A list of expressions. All expressions in the array must have the same shape. The list must contain at least one element.

Return (*Expression*)

`Expr.condense`

```
Expr.condense(Expression e) -> Expression
```

Flatten expression and remove all structural zeros.

Parameters `e` (*Expression*) – Expression to be condensed.

Return (*Expression*)

`Expr.constTerm`

```
Expr.constTerm(float[] vals1) -> Expression
Expr.constTerm(float[][] vals2) -> Expression
Expr.constTerm(int size, float val) -> Expression
Expr.constTerm(int[] shp, float val) -> Expression
Expr.constTerm(int[] shp, int[][] sparsity, float[] val) -> Expression
Expr.constTerm(int[] shp, int[][] sparsity, float val) -> Expression
Expr.constTerm(float val) -> Expression
Expr.constTerm(Matrix m) -> Expression
Expr.constTerm(NDSparseArray nda) -> Expression
```

Create an expression consisting of a constant vector of values.

Parameters

- `vals1` (`float[]`) – A vector initializing the expression.
- `vals2` (`float[][]`) – A two-dimensional array initializing the expression.
- `size` (`int`) – Length of the vector to be constructed.
- `val` (`float`) – A scalar value to be repeated in all entries of the expression.
- `val` (`float[]`) – A scalar value to be repeated in all entries of the expression.
- `shp` (`int[]`) – Defines the shape of the expression.
- `sparsity` (`int[][]`) – Sparsity pattern.
- `m` (*Matrix*) – A Matrix of values initializing the expression.
- `nda` (*NDSparseArray*) – An multi-dimensional sparse array initializing the expression.

Return (*Expression*)

`Expr.dot`

```
Expr.dot(float[] c1, Expression e) -> Expression
Expr.dot(float[][] c2, Expression e) -> Expression
Expr.dot(NDSparseArray nda, Expression e) -> Expression
Expr.dot(Matrix m, Expression e) -> Expression
Expr.dot(Expression e, float[] c1) -> Expression
Expr.dot(Expression e, NDSparseArray nda) -> Expression
Expr.dot(Expression e, float[][] c2) -> Expression
Expr.dot(Expression e, Matrix m) -> Expression
```

Return an object representing the inner product (dot product) $x^T y = \sum_{i=1}^{n} x_i y_i$ of two objects x, y of size n.

Both arguments must have the same length when flattened. In particular, they can be two vectors of the same length or two matrices of the same shape.

Parameters

- `c1` (`float[]`) – A one-dimensional coefficient vector.
- `e` (*Expression*) – An expression object.
- `c2` (`float[][]`) – A two-dimensional coefficient array.
- `nda` (*NDSparseArray*) – A multi-dimensional sparse array.
- `m` (*Matrix*) – A matrix object.

Return (*Expression*)

`Expr.eval`

```
eval(WorkStack rs, WorkStack ws, WorkStack xs)
eval() -> FlatExpr
```

Evaluate the expression and push the result onto the `rs` work stack.

Parameters

- `rs` (*WorkStack*) – The stack where the result of the evaluation is stored.
- `ws` (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- `xs` (*WorkStack*)

Return (*FlatExpr*)

`Expr.flatten`

```
Expr.flatten(Expression e) -> Expression
```

Reshape the expression into a vector.

Parameters `e` (*Expression*) – The expression to be flattened.

Return (*Expression*)

`Expr.getND`

```
getND() -> int
```

Return the number of dimensions in the expression.

Return (int)

`Expr.getShape`

```
getShape() -> int[]
```

Return the shape of the expression

Return (int[])

`Expr.hstack`

```
Expr.hstack(Expression[] exprs) -> Expression
Expr.hstack(Expression e1, Expression e2) -> Expression
Expr.hstack(Expression e1, float a2) -> Expression
Expr.hstack(float a1, Expression e2) -> Expression
Expr.hstack(float a1, float a2, Expression e3) -> Expression
Expr.hstack(float a1, Expression e2, float a3) -> Expression
Expr.hstack(float a1, Expression e2, Expression e3) -> Expression
Expr.hstack(Expression e1, float a2, float a3) -> Expression
Expr.hstack(Expression e1, float a2, Expression e3) -> Expression
Expr.hstack(Expression e1, Expression e2, float a3) -> Expression
Expr.hstack(Expression e1, Expression e2, Expression e3) -> Expression
```

Stack a list of expressions horizontally (i.e. along the second dimension). The expressions must have the same shape, except for the second dimension. The arguments may be any combination of expressions, scalar constants and variables.

For example, if x^1, x^2, x^3 are three vectors of length n then their horizontal stack is the matrix

$$\begin{bmatrix} | & | & | \\ x^1 & x^2 & x^3 \\ | & | & | \end{bmatrix}$$

of shape `(n,3)`.

Parameters

- `exprs` (*Expression*[]) – A list of expressions.
- `e1` (*Expression*) – An expression.
- `e2` (*Expression*) – An expression.
- `a2` (`float`) – A scalar constant.
- `a1` (`float`) – A scalar constant.
- `e3` (*Expression*) – An expression.
- `a3` (`float`) – A scalar constant.

Return (*Expression*)

`Expr.mul`

```
Expr.mul(Matrix mx, Variable v) -> Expression
Expr.mul(Variable v, Matrix mx) -> Expression
Expr.mul(Matrix mx, Expression e) -> Expression
Expr.mul(Expression e, Matrix mx) -> Expression
Expr.mul(float[][] a, Expression e) -> Expression
Expr.mul(Expression e, float[][] a) -> Expression
Expr.mul(float[] a, Expression e) -> Expression
Expr.mul(Expression e, float[] a) -> Expression
Expr.mul(float c, Expression e) -> Expression
Expr.mul(Expression e, float c) -> Expression
```

Compute the product (in the sense of matrix multiplication or scalar-by-matrix multiplication) of two arguments.

The operands must be at most two-dimensional. One of the arguments must be a constant, a vector of constants or a matrix of constants. The other argument can be a variable or expression. This allows to produce matrix expressions where the entries are linear combinations of variables.

The size and shape of the arguments must adhere to the rules of linear algebra.

Parameters
- `mx` (*Matrix*) – A matrix.
- `v` (*Variable*) – A variable.
- `e` (*Expression*) – An expression.
- `a` (`float`[][]) – Scalar data.
- `a` (`float`[]) – Scalar data.
- `c` (`float`) – A scalar value.

Return (*Expression*)

`Expr.mulDiag`

```
Expr.mulDiag(float[][] a, Expression expr) -> Expression
Expr.mulDiag(float[][] a, Variable v) -> Expression
Expr.mulDiag(Expression expr, float[][] a) -> Expression
Expr.mulDiag(Variable v, float[][] a) -> Expression
Expr.mulDiag(Matrix mx, Expression expr) -> Expression
Expr.mulDiag(Expression expr, Matrix mx) -> Expression
Expr.mulDiag(Matrix mx, Variable v) -> Expression
Expr.mulDiag(Variable v, Matrix mx) -> Expression
```

Compute the diagonal of the product of two matrices. If $A \in \mathbb{M}(m, n)$ and $B \in \mathbb{M}(n, p)$, the result is a vector expression of length n equal to $\mathbf{diag}(AB)$.

Parameters
- `a` (`float`[][]) – A constant matrix.
- `expr` (*Expression*) – An expression object.
- `v` (*Variable*) – A variable object.
- `mx` (*Matrix*) – A matrix object.

Return (*Expression*)

`Expr.mulElm`

```
Expr.mulElm(Expression expr, NDSparseArray spm) -> Expression
Expr.mulElm(Expression expr, float[] a1) -> Expression
Expr.mulElm(Expression expr, float[][] a2) -> Expression
Expr.mulElm(Expression expr, Matrix m) -> Expression
```

(continues on next page)

(continued from previous page)

```
Expr.mulElm(float[] a1, Expression expr) -> Expression
Expr.mulElm(float[][] a2, Expression expr) -> Expression
Expr.mulElm(NDSparseArray spm, Expression expr) -> Expression
Expr.mulElm(Matrix m, Expression expr) -> Expression
```

Returns the element-wise product of two items. The two operands must have the same shape and the returned expression also has this shape.

Parameters
- `expr` (*Expression*) – An expression object.
- `spm` (*NDSparseArray*) – A multidimensional sparse array object.
- `a1` (`float`[]) – A one-dimensional coefficient array.
- `a2` (`float`[][]) – A two-dimensional coefficient array.
- `m` (*Matrix*) – A matrix object.

Return (*Expression*)

`Expr.neg`

```
Expr.neg(Expression e) -> Expression
```

Return a new expression object representing the given one with opposite sign.

Parameters `e` (*Expression*) – An expression object.

Return (*Expression*)

`Expr.numNonzeros`

```
numNonzeros() -> int
```

Return the number of non-zero elements in the expression.

Return (`int`)

`Expr.ones`

```
Expr.ones(int size) -> Expression
Expr.ones(int[] shp) -> Expression
Expr.ones(int[] shp, int[][] sparsity) -> Expression
Expr.ones() -> Expression
```

Create an expression consisting of ones.

Parameters
- `size` (`int`) – Length of the vector to be constructed.
- `shp` (`int`[]) – Defines the shape of the expression.
- `sparsity` (`int`[][]) – Defines the sparsity pattern of the expression - everything outside the sparsitry patterm will be zero.

Return (*Expression*)

`Expr.outer`

```
Expr.outer(Variable v, float[] a) -> Expression
Expr.outer(float[] a, Variable v) -> Expression
Expr.outer(Variable v, Matrix m) -> Expression
Expr.outer(Matrix m, Variable v) -> Expression
Expr.outer(Expression e, float[] a) -> Expression
Expr.outer(float[] a, Expression e) -> Expression
```

Return an expression representing the outer product xy^T of two vectors x, y. If x has length `k` and y has length `n` then the result is of shape `(k,n)`.

Parameters

- `v` (*Variable*) – A vector variable.
- `a` (`float[]`) – A vector of constants.
- `m` (*Matrix*) – A one-dimensional matrix.
- `e` (*Expression*) – A vector expression.

Return (*Expression*)

`Expr.repeat`

```
Expr.repeat(Expression e, int n, int d) -> Expression
```

Repeat an expression a number of times in the given dimension. This is equivalent to stacking n copies of the expression in dimension d; see *Expr.stack*.

Parameters

- `e` (*Expression*) – The expression to repeat.
- `n` (`int`) – Number of times to repeat. Must be strictly positive.
- `d` (`int`) – The dimension in which to repeat. Must define a valid dimension index.

Return (*Expression*)

`Expr.reshape`

```
Expr.reshape(Expression e, int[] newshape) -> Expression
Expr.reshape(Expression e, int size) -> Expression
Expr.reshape(Expression e, int dimi, int dimj) -> Expression
```

Reshape the expression into a different shape with the same number of elements.

Parameters

- `e` (*Expression*) – The expression to reshape.
- `newshape` (`int[]`) – Reshape into an expression of this shape.
- `size` (`int`) – Reshape into a one-dimensional expression of this size.
- `dimi` (`int`) – The first dimension size.
- `dimj` (`int`) – The second dimension size.

Return (*Expression*)

`Expr.size`

```
size() -> int
```

Return the expression size, i.e. the product of the lengths along each dimension.

Return (`int`)

`Expr.stack`

```
Expr.stack(int dim, Expression[] exprs) -> Expression
Expr.stack(int dim, Expression e1, Expression e2) -> Expression
Expr.stack(int dim, Expression e1, float a2) -> Expression
Expr.stack(int dim, float a1, Expression e2) -> Expression
Expr.stack(int dim, float a1, float a2, Expression e1) -> Expression
Expr.stack(int dim, float a1, Expression e2, float a3) -> Expression
```

(continues on next page)

(continued from previous page)

```
Expr.stack(int dim, float a1, Expression e2, Expression e3) -> Expression
Expr.stack(int dim, Expression e1, float a2, float a3) -> Expression
Expr.stack(int dim, Expression e1, float a2, Expression e3) -> Expression
Expr.stack(int dim, Expression e1, Expression e2, float a3) -> Expression
Expr.stack(int dim, Expression e1, Expression e2, Expression e3) -> Expression
Expr.stack(Expression[][] exprs) -> Expression
```

Stack a list of expressions along an arbitrary dimension. All expressions must have the same shape, except for dimension `dim`. The arguments may be any combination of expressions, scalar constants and variables.

For example, suppose A, B are two $n \times m$ matrices. Then stacking them in the first dimension produces a matrix of shape `(2n,m)`:

$$\left[\begin{array}{c} A \\ B \end{array}\right],$$

stacking them in the second dimension produces a matrix of shape `(n,2m)`:

$$\left[\begin{array}{cc} A & B \end{array}\right],$$

and stacking them in the third dimension produces a three-dimensional array of shape `(n,m,2)`.

The version which takes a two-dimensional array of expressions constructs a block matrix with the given expressions as blocks. The dimensions of the blocks must be suitably compatible.

Parameters

- `dim` (`int`) – The dimension in which to stack.
- `exprs` (*Expression*[]) – A list of expressions.
- `exprs` (*Expression*[][]) – A list of expressions.
- `e1` (*Expression*) – An expression.
- `e2` (*Expression*) – An expression.
- `a2` (`float`) – A scalar constant.
- `a1` (`float`) – A scalar constant.
- `a3` (`float`) – A scalar constant.
- `e3` (*Expression*) – An expression.

Return (*Expression*)

`Expr.sub`

```
Expr.sub(Expression e1, Expression e2) -> Expression
Expr.sub(Expression e1, float[] a1) -> Expression
Expr.sub(Expression e1, float[][] a2) -> Expression
Expr.sub(float[] a1, Expression e2) -> Expression
Expr.sub(float[][] a2, Expression e2) -> Expression
Expr.sub(Expression e1, float c) -> Expression
Expr.sub(float c, Expression e2) -> Expression
Expr.sub(Expression e1, Matrix m) -> Expression
Expr.sub(Matrix m, Expression e2) -> Expression
Expr.sub(Expression e1, NDSparseArray n) -> Expression
Expr.sub(NDSparseArray n, Expression e2) -> Expression
```

Computes the difference of two expressions. The expressions must have the same shape and the result will be also an expression of that shape. The allowed combinations of arguments are the same as for *Expr.add*.

Parameters

- `e1` (*Expression*) – An expression.
- `e2` (*Expression*) – An expression.

- a1 (float[]) – An array of constants.
- a2 (float[][]) – An array of constants.
- c (float) – A constant.
- m (*Matrix*) – A Matrix object.
- n (*NDSparseArray*) – An NDSparseArray object.

Return (*Expression*)

`Expr.sum`

```
Expr.sum(Expression expr) -> Expression
Expr.sum(Expression expr, int dim) -> Expression
```

Sum the elements of an expression. Without extra arguments, all elements are summed into a scalar expression of size 1.

With argument `dim`, elements are summed along a specific dimension, 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.

For example, if the argument is an $n \times m$ matrix then summing along the 0-th dimension computes the $1 \times m$ vector of column sums, and summing along the 1-st dimension computes the $n \times 1$ vector of row sums.

Parameters
- expr (*Expression*) – An expression object.
- dim (int) – The dimension along which to sum.

Return (*Expression*)

`Expr.transpose`

```
Expr.transpose(Expression e) -> Expression
```

Transpose a two-dimensional expression.

Parameters e (*Expression*) – Expression to transpose.

Return (*Expression*)

`Expr.vstack`

```
Expr.vstack(Expression[] exprs) -> Expression
Expr.vstack(Expression e1, Expression e2) -> Expression
Expr.vstack(Expression e1, float a2) -> Expression
Expr.vstack(float a1, Expression e2) -> Expression
Expr.vstack(Expression e1, Expression e2, Expression e3) -> Expression
Expr.vstack(Expression e1, Expression e2, float a3) -> Expression
Expr.vstack(Expression e1, float a2, Expression e3) -> Expression
Expr.vstack(Expression e1, float a2, float a3) -> Expression
Expr.vstack(float a1, Expression e2, Expression e3) -> Expression
Expr.vstack(float a1, Expression e2, float a3) -> Expression
Expr.vstack(float a1, float a2, Expression e3) -> Expression
Expr.vstack(float a1, float a2, float a3) -> Expression
```

Stack a list of expressions vertically (i.e. along the first dimension). The expressions must have the same shape, except for the first dimension. The arguments may be any combination of expressions, scalar constants and variables.

For example, if y^1, y^2, y^3 are three horizontal vectors of length n (and shape `(1,n)`) then their vertical stack is the matrix

$$\begin{bmatrix} -y^1- \\ -y^2- \\ -y^3- \end{bmatrix}$$

of shape (3,n).

Parameters

- exprs (*Expression*[]) – A list of expressions.
- e1 (*Expression*) – An expression.
- e2 (*Expression*) – An expression.
- a2 (float) – A scalar constant.
- a1 (float) – A scalar constant.
- e3 (*Expression*) – An expression.
- a3 (float) – A scalar constant.

Return (*Expression*)

Expr.zeros

```
Expr.zeros(int size) -> Expression
Expr.zeros(int[] shp) -> Expression
```

Create an expression consisting of zeros.

Parameters

- size (int) – Length of the vector to be constructed.
- shp (int[]) – Defines the shape of the expression.

Return (*Expression*)

14.2.9 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 and push the result onto the work stack.

Expression.getDim – Return the d'th dimension in the expression.

Expression.getND – Return the number of dimensions in the expression.

Expression.getShape – Get the shape of the expression.

Expression.getSize – Return the total number of elements in the expression (the product of the dimensions).

Expression.index – Get a single element in the expression.

Expression.pick – Pick a number of elements from the expression.

Expression.slice – Get a slice of the expression.

Expression.toString – Return a string representation of the expression object.

Implemented by *BaseExpression*, *Variable*

Expression.eval

```
eval() -> FlatExpr
eval(WorkStack rs, WorkStack ws, WorkStack xs)
```

Evaluate the expression and push the result onto the rs work stack.

Parameters

- rs (*WorkStack*) – The stack where the result of the evaluation is stored.
- ws (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- xs (*WorkStack*)

Return (*FlatExpr*)

`Expression.getDim`

```
getDim(int d) -> int
```

Return the d'th dimension in the expression.

Parameters `d` (`int`)
Return (`int`)

`Expression.getND`

```
getND() -> int
```

Return the number of dimensions in the expression.

Return (`int`)

`Expression.getShape`

```
getShape() -> int[]
```

Get the shape of the expression.

Return (`int`[])

`Expression.getSize`

```
getSize() -> int
```

Return the total number of elements in the expression (the product of the dimensions).

Return (`int`)

`Expression.index`

```
index(int i) -> Expression
index(int[] indexes) -> Expression
```

Get a single element in the expression.

Parameters
- `i` (`int`) – Index of the element to pick.
- `indexes` (`int`[]) – Multi-dimensional index of the element to pick.

Return (*Expression*)

`Expression.pick`

```
pick(int[] indexes) -> Expression
pick(int[][] indexrows) -> Expression
```

Picks a number of elements from the expression and returns them as a one-dimensional expression.

Parameters
- `indexes` (`int`[]) – Indexes of the elements to pick
- `indexrows` (`int`[][]) – Indexes of the elements to pick. Each row defines a separate multi-dimensional index.

Return (*Expression*)

Expression.slice

```
slice(int first, int last) -> Expression
slice(int[] firsta, int[] lasta) -> Expression
```

Get a slice of the expression.

Parameters

- `first` (`int`) – Index of the first element in the slice.
- `last` (`int`) – Index of the last element in the slice plus one.
- `firsta` (`int`[]) – Multi-dimensional index of the first element in the slice.
- `lasta` (`int`[]) – Multi-dimensional index of the element after the end of the slice.

Return (*Expression*)

Expression.toString

```
toString() -> str
```

Return a string representation of the expression object.

Return (`str`)

14.2.10 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 string representation of the expression.

FlatExpr.size

```
size() -> int
```

Get the number of non-zero elements in the expression.

Return (`int`)

FlatExpr.toString

```
toString() -> str
```

Create a human readable string representation of the expression.

Return (`str`)

14.2.11 Class LinPSDDomain

mosek.fusion.LinPSDDomain

Represent a linear PSD domain.

14.2.12 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.

Implements *ModelConstraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.slice – Create a slice constraint.

Constraint.update – Update part of a constraint.

LinearConstraint.toString – Create a human readable string representation of the constraint.

LinearConstraint.toString

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

14.2.13 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

LinearDomain.withShape – Set the shape of the domain.

LinearDomain.integral

```
integral() -> LinearDomain
```

Modify a given domain restricting its elements to be integral.

Return (*LinearDomain*)

LinearDomain.sparse

```
sparse() -> LinearDomain
sparse(int[] sparsity) -> LinearDomain
sparse(int[][] sparsity) -> LinearDomain
```

Creates a domain exploiting sparsity.

Parameters

- sparsity (int[])
- sparsity (int[][])

Return (*LinearDomain*)

LinearDomain.symmetric

```
symmetric() -> SymmetricLinearDomain
```

Creates a symmetric domain

Return (*SymmetricLinearDomain*)

LinearDomain.withShape

```
withShape(int[] shp) -> LinearDomain
```

Set the shape of the domain.

Parameters shp (int[]) – The shape of the domain

Return (*LinearDomain*)

14.2.14 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

Implements *ModelConstraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.slice – Create a slice constraint.

Constraint.update – Update part of a constraint.

ModelConstraint.toString – Create a human readable string representation of the constraint.

14.2.15 Class LinearPSDVariable

mosek.fusion.LinearPSDVariable

This class represents a positive semidefinite variable.

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this variable
BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension
BaseVariable.transpose – Transpose a vector or matrix variable
LinearPSDVariable.toString – Create a string representation of the variable.

`LinearPSDVariable.toString`

```
toString() -> str
```

Create a string representation of the variable.

Return (str)

14.2.16 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.

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this variable
BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension
BaseVariable.transpose – Transpose a vector or matrix variable
LinearVariable.toString – Create a string representation of the variable.

`LinearVariable.toString`

```
toString() -> str
```

Create a string representation of the variable.

Return (str)

14.2.17 Class Matrix

`mosek.fusion.Matrix`

Base class for all matrix objects. It can be used to create and manipulate matrices of constant coefficients both in dense and sparse format. To operate with matrices containing variables and linear expressions use the classes *Expr* and *Variable*.

Members *Matrix.get* – Get a single entry.
Matrix.getDataAsArray – Return a dense array of values.
Matrix.getDataAsTriplets – Return the matrix data in sparse triplet format.
Matrix.isSparse – Returns true if the matrix is sparse.
Matrix.numColumns – Returns the number of columns in the matrix.
Matrix.numNonzeros – Returns the number of non-zeros in the matrix.
Matrix.numRows – Returns the number of 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 dense matrix from the given data.
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 matrix from the given data.

`Matrix.antidiag`

```
Matrix.antidiag(float[] d) -> Matrix
Matrix.antidiag(float[] d, int k) -> Matrix
Matrix.antidiag(int n, float val) -> Matrix
Matrix.antidiag(int n, float val, int k) -> Matrix
```

Create a sparse square matrix with a given vector as anti-diagonal.

Parameters

- d (float[]) – The anti-diagonal vector.
- k (int) – The anti-diagonal index. $k = 0$ is the default and means the main anti-diagonal. $k > 0$ means above, and $k < 0$ means below the main anti-diagonal.
- n (int) – The dimension of the matrix.
- val (float) – Use this value for all anti-diagonal elements.

Return (*Matrix*)

`Matrix.dense`

```
Matrix.dense(float[][] data) -> Matrix
Matrix.dense(int dimi, int dimj, float[] data) -> Matrix
Matrix.dense(int dimi, int dimj, float value) -> Matrix
Matrix.dense(Matrix other) -> Matrix
```

Create a dense matrix from the given data.

Parameters

- `data` (`float`[][]) – A one- or two-dimensional array of matrix coefficients.
- `data` (`float`[]) – A one- or two-dimensional array of matrix coefficients.
- `dimi` (`int`) – Number of rows.
- `dimj` (`int`) – Number of columns.
- `value` (`float`) – Use this value for all elements.
- `other` (*`Matrix`*) – Create a dense matrix from another matrix.

Return (*`Matrix`*)

`Matrix.diag`

```
Matrix.diag(float[] d) -> Matrix
Matrix.diag(float[] d, int k) -> Matrix
Matrix.diag(int n, float val) -> Matrix
Matrix.diag(int n, float val, int k) -> Matrix
Matrix.diag(Matrix[] md) -> Matrix
Matrix.diag(int num, Matrix mv) -> Matrix
```

Create a sparse square matrix with a given vector as diagonal.

Parameters

- `d` (`float`[]) – The diagonal vector.
- `k` (`int`) – The diagonal index. $k = 0$ is the default and means the main diagonal. $k > 0$ means above, and $k < 0$ means below the main diagonal.
- `n` (`int`) – The dimension of the matrix.
- `val` (`float`) – Use this value for all diagonal elements.
- `md` (*`Matrix`*[]) – A list of square matrices that are used to create a block-diagonal square matrix.
- `num` (`int`) – Number of times to repeat the `mv` matrix.
- `mv` (*`Matrix`*) – A matrix to be repeated in all blocks of a block-diagonal square matrix.

Return (*`Matrix`*)

`Matrix.eye`

```
Matrix.eye(int n) -> Matrix
```

Construct the identity matrix of size n.

Parameters `n` (`int`) – The dimension of the matrix.

Return (*`Matrix`*)

`Matrix.get`

```
get(int i, int j) -> float
```

Get a single entry.

Parameters
- `i` (`int`) – Row index.
- `j` (`int`) – Column index.

Return (`float`)

`Matrix.getDataAsArray`

```
getDataAsArray() -> float[]
```

Return the matrix elements as a dense array in row-major format.

Return (`float`[])

`Matrix.getDataAsTriplets`

```
getDataAsTriplets(int[] subi, int[] subj, float[] val)
```

Return the matrix data in sparse triplet format. Data is copied to the arrays `subi`, `subj` and `val` which must be pre-allocated to hold at least the number of non-zeros in the matrix.

The data returned must be ordered with `subi` as primary key and `subj` as secondary key.

Parameters
- `subi` (`int`[]) – Row subscripts are returned in this array.
- `subj` (`int`[]) – Column subscripts are returned in this array.
- `val` (`float`[]) – Coefficient values are returned in this array.

`Matrix.isSparse`

```
isSparse() -> bool
```

Returns true if the matrix is sparse.

Return (`bool`)

`Matrix.numColumns`

```
numColumns() -> int
```

Returns the number of columns in the matrix.

Return (`int`)

`Matrix.numNonzeros`

```
numNonzeros() -> int
```

Returns the number of non-zeros in the matrix.

Return (`int`)

`Matrix.numRows`

```
numRows() -> int
```

Returns the number of rows in the matrix.

Return (`int`)

`Matrix.ones`

```
Matrix.ones(int n, int m) -> Matrix
```

Construct a matrix filled with ones.

Parameters
- `n` (`int`) – Number of rows.
- `m` (`int`) – Number of columns.

Return (*Matrix*)

`Matrix.sparse`

```
Matrix.sparse(int nrow, int ncol, int[] subi, int[] subj, float[] val) -> Matrix
Matrix.sparse(int[] subi, int[] subj, float[] val) -> Matrix
Matrix.sparse(int[] subi, int[] subj, float val) -> Matrix
Matrix.sparse(int nrow, int ncol, int[] subi, int[] subj, float val) -> Matrix
Matrix.sparse(int nrow, int ncol) -> Matrix
Matrix.sparse(float[][] data) -> Matrix
Matrix.sparse(Matrix[][] blocks) -> Matrix
Matrix.sparse(Matrix mx) -> Matrix
```

Create a sparse matrix from the given data.

Parameters
- `nrow` (`int`) – Number of rows.
- `ncol` (`int`) – Number of columns.
- `subi` (`int`[]) – Row subscripts of non-zero elements.
- `subj` (`int`[]) – Column subscripts of non-zero elements.
- `val` (`float`[]) – Coefficients of non-zero elements.
- `val` (`float`) – Coefficients of non-zero elements.
- `data` (`float`[][]) – Dense data array.
- `blocks` (*Matrix*[][]) – The matrix data in block format. 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 (*Matrix*)

`Matrix.toString`

```
toString() -> str
```

Get a string representation of the matrix.

Return (str)

`Matrix.transpose`

```
transpose() -> Matrix
```

Transpose the matrix.

Return (*Matrix*)

14.2.18 Class Model

`mosek.fusion.Model`

The object containing all data related to a single optimization model.

Implements `BaseModel`

Members *Model.acceptedSolutionStatus* – Set the accepted solution status.
Model.breakSolver – Request that the solver terminates as soon as possible.
Model.clone – Return a copy of the model.
Model.constraint – Create a new constraint in the model.
Model.dispose – Destroy the Model object
Model.dualObjValue – Get the dual objective value in the current solution.
Model.flushSolutions – If any solution values have been provided, flush those values to the underlying task.
Model.getAcceptedSolutionStatus – Get the accepted solution status.
Model.getConstraint – Get the constraint matching the given name or linear index.
Model.getDualSolutionStatus – Return the status of the dual solution.
Model.getName – Return the model name, or an empty string if it has not been set.
Model.getPrimalSolutionStatus – Return the status of the primal solution.
Model.getProblemStatus – Return the status of the problem.
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 matching the given name or linear index.
Model.hasConstraint – Check whether the model contains a constraint with a given name.
Model.hasVariable – Check whether the model contains a variable with a given name.
Model.objective – Replace the objective expression.
Model.primalObjValue – Get the primal objective value in the current solution.
Model.selectedSolution – Chooses a solution.
Model.setCallbackHandler – Attach a progress callback handler.
Model.setDataCallbackHandler – Attach a data callback handler.
Model.setLogHandler – Attach a log handler.
Model.setSolverParam – Set a solver parameter
Model.solve – Attempt to optimize the model.
Model.updateObjective – Update part of the objective.
Model.variable – Create a new variable in the model.
Model.writeTask – Dump the current solver task to a file.

Static members *Model.getVersion* – Return MOSEK version.
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.

`Model.acceptedSolutionStatus`

```
acceptedSolutionStatus(AccSolutionStatus what)
```

Set the accepted solution status. This defines which solution status values are considered as *acceptable* when fetching a solution. Requesting a solution value for a variable or constraint when the status does not match at least the accepted value will cause an error.

By default the accepted solution status is *AccSolutionStatus.Optimal*. It is necessary to change the accepted status to access sub-optimal solutions and infeasibility certificates.

The methods *Model.getPrimalSolutionStatus* and *Model.getDualSolutionStatus* can be used to get the *actual* status of the solutions.

Parameters `what` (*AccSolutionStatus*) – The new accepted solution status.

Model.breakSolver

```
breakSolver()
```

Request that the solver terminates as soon as possible. This must be called from another thread than the one in which `solve()` was called, or from a callback function.

The method does not stop the solver directly, rather it sets a flag that the solver checks occasionally, indicating it should terminate.

Model.clone

```
clone() -> Model
```

Return a copy of the model.

Return (*Model*)

Model.constraint

```
constraint(str name, Expression expr, PSDDomain psddom) -> Constraint
constraint(Expression expr, PSDDomain psddom) -> Constraint
constraint(str name, Expression expr, LinPSDDomain lpsddom) -> Constraint
constraint(Expression expr, LinPSDDomain lpsddom) -> Constraint
constraint(str name, Expression expr, LinearDomain ldom) -> Constraint
constraint(Expression expr, LinearDomain ldom) -> Constraint
constraint(str name, Expression expr, RangeDomain rdom) -> Constraint
constraint(Expression expr, RangeDomain rdom) -> Constraint
constraint(str name, Expression expr, ConeDomain qdom) -> Constraint
constraint(Expression expr, ConeDomain qdom) -> Constraint
```

Adds a new constraint to the model. A constraint is always a statement that *an expression or variable* belongs to *a domain*. Constraints can have optional names.

Typical domains used for defining constraints include:

- *Domain.lessThan*, *Domain.greaterThan*, *Domain.inRange*, *Domain.equalsTo* — puts linear bounds $E \leq u$, $l \leq E$, $l \leq E \leq u$ or $E = c$ on an expresion E.
- *Domain.inQCone*, *Domain.inRotatedQCone* — constrains a vector or matrix expression E to a second-order cone.
- *Domain.inPExpCone*, *Domain.inPPowerCone* — constrains a vector or matrix expression E to an exponential or power cone.
- *Domain.inPSDCone* — constrains a square matrix expression E to be positive semidefinite.

See *Domain* for a full list of domains.

Parameters

- `name` (`str`) – 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.
- `psddom` (*PSDDomain*) – A positive semidefinte domain.
- `lpsddom` (*LinPSDDomain*) – A linear positive semidefinte domain.
- `ldom` (*LinearDomain*) – A linear domain.
- `rdom` (*RangeDomain*) – A ranged domain.
- `qdom` (*ConeDomain*) – A domain in a cone.

Return (*Constraint*)

Model.dispose

```
dispose()
```

Destroy the Model object. This removes all references to other objects from the Model.

This helps garbage collection by removing cyclic references, and in some cases it is necessary to ensure that the garbage collector can collect the Model object and assosiated objects.

`Model.dualObjValue`

```
dualObjValue() -> float
```

Get the dual objective value in the current solution.

Return (`float`)

`Model.flushSolutions`

```
flushSolutions()
```

If any solution values have been provided, flush those values to the underlying task.

`Model.getAcceptedSolutionStatus`

```
getAcceptedSolutionStatus() -> AccSolutionStatus
```

Get the accepted solution status.

Return (*`AccSolutionStatus`*)

`Model.getConstraint`

```
getConstraint(str name) -> Constraint
getConstraint(int index) -> Constraint
```

Get the constraint matching the given name or linear index. Constraints are assigned indices in the order they are added to the model.

Parameters

- `name` (`str`) – The constraint's name.
- `index` (`int`) – The constraint's linear index.

Return (*`Constraint`*)

`Model.getDualSolutionStatus`

```
getDualSolutionStatus(SolutionType which) -> SolutionStatus
getDualSolutionStatus() -> SolutionStatus
```

Return the status of the dual solution. If no solution type is given the solution set with *`Model.selectedSolution`* is checked. It is recommended to check the problem and solution status before accessing the solution values.

Parameters `which` (*`SolutionType`*) – The type of the solution for which status is requested.

Return (*`SolutionStatus`*)

`Model.getName`

```
getName() -> str
```

Return the model name, or an empty string if it has not been set.

Return (str)

Model.getPrimalSolutionStatus

```
getPrimalSolutionStatus(SolutionType which) -> SolutionStatus
getPrimalSolutionStatus() -> SolutionStatus
```

Return the status of the primal solution. If no solution type is given the solution set with *Model.selectedSolution* is checked. It is recommended to check the problem and solution status before accessing the solution values.

Parameters which (*SolutionType*) – The type of the solution for which status is requested.

Return (*SolutionStatus*)

Model.getProblemStatus

```
getProblemStatus(SolutionType which) -> ProblemStatus
getProblemStatus() -> ProblemStatus
```

Return the status of the problem. If no solution type is given the solution set with *Model.selectedSolution* is checked. It is recommended to check the problem and solution status before accessing the solution values.

Parameters which (*SolutionType*) – The type of the solution.

Return (*ProblemStatus*)

Model.getSolverDoubleInfo

```
getSolverDoubleInfo(str name) -> float
```

This method returns the value for the specified double solver information item. The information items become available during and after the solver execution. A runtime exception is thrown if a non-existing information item is requested. The double information items are listed in Section *Double information items*.

Parameters name (str) – A string name of the information item.

Return (float)

Model.getSolverIntInfo

```
getSolverIntInfo(str name) -> int
```

This method returns the value for the specified integer solver information item. The information items become available during and after the solver execution. A runtime exception is thrown if a non-existing information item is requested. The integer information items are listed in Section *Integer information items.*.

Parameters name (str) – A string name of the information item.

Return (int)

Model.getSolverLIntInfo

```
getSolverLIntInfo(str name) -> int
```

This method returns the value for the specified long solver information item. The information items become available during and after the solver execution. A runtime exception is thrown if a non-existing information item is requested. The long integer information items are listed in Section *Long integer information items.*.

Parameters `name` (`str`) – A string name of the information item.

Return (`int`)

`Model.getTask`

```
getTask() -> mosek.Task
```

Returns the underlying **MOSEK** Task object. Note that the returned object is the actual underlying object, not a copy. This means if the returned object is modified by the user, the *`Model`* object may become invalid. Accessing the task object should never be necessary, except maybe for advanced debugging. For details on the Task object see the Optimizer API documentation.

Return (`Task`)

`Model.getVariable`

```
getVariable(str name) -> Variable
getVariable(int index) -> Variable
```

Get the variable matching the given name or linear index. Variables are assigned indices in the order they are added to the model.

Parameters

- `name` (`str`) – The variable's name.
- `index` (`int`) – The variable's linear index.

Return (*`Variable`*)

`Model.getVersion`

```
Model.getVersion() -> str
```

Returns the **MOSEK** version as a string, for example "9.0.98".

Return (`str`)

`Model.hasConstraint`

```
hasConstraint(str name) -> bool
```

Check whether the model contains a constraint with a given name.

Parameters `name` (`str`) – The constraint name.

Return (`bool`)

`Model.hasVariable`

```
hasVariable(str name) -> bool
```

Check whether the model contains a variable with a given name.

Parameters `name` (`str`) – The variable name.

Return (`bool`)

`Model.objective`

```
objective(str name, ObjectiveSense sense, Expression expr)
objective(str name, ObjectiveSense sense, float c)
objective(str name, float c)
objective(ObjectiveSense sense, Expression expr)
objective(ObjectiveSense sense, float c)
objective(float c)
```

Replace the objective expression. This method must be called at least once before the first *Model.solve*.

Parameters

- name (str) – Name of the objective. This may be any string, and it has no function except when writing the problem to an external file format.
- sense (*ObjectiveSense*) – The objective sense. Defines whether the objective must be minimized or maximized.
- expr (*Expression*) – The objective expression. This must be an expression that evaluates to a scalar.
- c (float) – A constant scalar.

Model.primalObjValue

```
primalObjValue() -> float
```

Get the primal objective value in the current solution.

Return (float)

Model.putlicensecode

```
Model.putlicensecode(int[] code)
```

Set the license code in the global environment.

Parameters code (int[])

Model.putlicensepath

```
Model.putlicensepath(str licfile)
```

Set the license path in the global environment.

Parameters licfile (str)

Model.putlicensewait

```
Model.putlicensewait(bool wait)
```

Set the license wait flag in the global environment. If set, **MOSEK** will wait until a license becomes available.

Parameters wait (bool)

Model.selectedSolution

```
selectedSolution(SolutionType soltype)
```

Chooses a solution. The values of variables and constraints will be read from the chosen solution. The default is to consider all solution types in the order of *SolutionType.Default*.

Parameters soltype (*SolutionType*)

Model.setCallbackHandler

```
setCallbackHandler(System.CallbackHandler h)
```

Attach a progress callback handler. During optimization this handler will be called, providing a code with the current state of the solver. Passing NULL detaches the current handler. See Section *Progress and data callback* for details and examples and the Optimizer API for information about callback codes.

The progress callback handler is a function of type `(mosek.callback) -> int`.

Parameters h (CallbackHandler) – The callback handler or NULL.

Model.setDataCallbackHandler

```
setDataCallbackHandler(System.DataCallbackHandler h)
```

Attach a data callback handler. During optimization this handler will be called, providing various information about the current state of the solution and solver. Passing NULL detaches the current handler. See Section *Progress and data callback* for details and examples and the Optimizer API for information about callback codes.

The data callback handler is a function of type `(mosek.callback, double[], int[], long[]) -> int`.

Parameters h (DataCallbackHandler) – The callback handler or NULL.

Model.setLogHandler

```
setLogHandler(System.StreamWriter h)
```

Attach a log handler. The solver log information will be sent to the stream handler. Passing NULL detaches the current handler.

The log handler must have a `write` method accepting a string, for example

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

Parameters h (StreamWriter) – The log handler object or NULL.

Model.setSolverParam

```
setSolverParam(str name, str strval)
setSolverParam(str name, int intval)
setSolverParam(str name, float floatval)
```

Set a solver parameter. Solver parameter values can be either symbolic values, integers or doubles, depending on the parameter. The value is automatically converted to a suitable type whenever possible. If this fails, an exception will be thrown. For example, if the parameter accepts a double value and is given a string, the string will be parsed to produce a double.

See Section *Parameters (alphabetical list sorted by type)* for a listing of all parameter settings.

Parameters

- name (str) – Name of the parameter to set
- strval (str) – A string value to assign to the parameter.
- intval (int) – An integer value to assign to the parameter.
- floatval (float) – A floating point value to assign to the parameter.

Model.solve

```
solve()
```

This calls the **MOSEK** solver to solve the problem defined in the model.

If no error occurs, on exit a solution status will be defined for the primal and the dual solutions. These can be obtained with *Model.getPrimalSolutionStatus* and *Model.getDualSolutionStatus*. Depending on the solution status, various values may be defined:

- If the model is primal-dual feasible, or nearly so, and the solver found a solution, the solution values can be accessed through the *Variable* and *Constraint* objects in the model. For integer problems only the primal solution is defined, while for continuous problems both primal and dual solutions are available.
- If the model is primal or dual infeasible, *only* the primal *or* the dual solution is defined, depending on the solution status. The available solution contains a certificate of infeasibility.
- If the status is unknown the solver ran into problems and did not find anything useful. In this case the solution values may be garbage.

The solution can be obtained with *Model.primalObjValue* and *Variable.level* and their dual analogues.

By default, trying to fetch a non-optimal solution using *Variable.level* or *Variable.dual* will cause an exception. To fetch infeasibility certificates or other less optimal solutions it is necessary to change the accepted solution flag with *Model.acceptedSolutionStatus*.

Model.updateObjective

```
updateObjective(Expression expr, Variable x)
```

Update the columns in the objective expression defined by x.

Parameters

- expr (*Expression*) – The expression to update with. This must have size 1.
- x (*Variable*) – The columns to replace.

Model.variable

```
variable(str name) -> Variable
variable(str name, int size) -> Variable
variable(str name, int size, LinearDomain ldom) -> Variable
variable(str name, int size, RangeDomain rdom) -> Variable
variable(str name, int size, ConeDomain qdom) -> Variable
variable(str name, int[] shp) -> Variable
variable(str name, int[] shp, LinearDomain ldom) -> Variable
variable(str name, int[] shp, RangeDomain rdom) -> Variable
variable(str name, int[] shp, ConeDomain qdom) -> Variable
variable(str name, LinearDomain ldom) -> Variable
variable(str name, RangeDomain rdom) -> Variable
variable(str name, ConeDomain qdom) -> Variable
variable() -> Variable
variable(int size) -> Variable
variable(int size, LinearDomain ldom) -> Variable
variable(int size, RangeDomain rdom) -> Variable
variable(int size, ConeDomain qdom) -> Variable
variable(int[] shp) -> Variable
variable(int[] shp, LinearDomain ldom) -> Variable
variable(int[] shp, RangeDomain rdom) -> Variable
variable(int[] shp, ConeDomain qdom) -> Variable
variable(LinearDomain ldom) -> Variable
```

(continues on next page)

(continued from previous page)

```
variable(RangeDomain rdom) -> Variable
variable(ConeDomain qdom) -> Variable
variable(str name, int[] shp, PSDDomain psddom) -> Variable
variable(str name, int n, PSDDomain psddom) -> Variable
variable(str name, int n, int m, PSDDomain psddom) -> Variable
variable(str name, PSDDomain psddom) -> Variable
variable(int n, PSDDomain psddom) -> Variable
variable(int n, int m, PSDDomain psddom) -> Variable
variable(PSDDomain psddom) -> Variable
variable(str name, int[] shp, LinPSDDomain lpsddom) -> Variable
variable(str name, int n, LinPSDDomain lpsddom) -> Variable
variable(str name, int n, int m, LinPSDDomain lpsddom) -> Variable
variable(str name, LinPSDDomain lpsddom) -> Variable
variable(int n, LinPSDDomain lpsddom) -> Variable
variable(int n, int m, LinPSDDomain lpsddom) -> Variable
variable(LinPSDDomain lpsddom) -> Variable
```

Create a new variable in the model. All variables must be created using this method. The many versions of the method accept a name (optional), the shape of the variable and its domain. The domain must be suitable for the given variable shape. If the domain is not provided, it is assumed that the variable is unbounded. If the dimension is not provided the variable is a single scalar variable.

Typical domains used for creating variables include:

- *Domain.lessThan*, *Domain.greaterThan*, *Domain.inRange* — creates a variable x with bounds $x \leq u$, $l \leq x$ or $l \leq x \leq u$.
- *Domain.inPSDCone* — creates a symmetric positive definite variable of dimension n.

Parameters

- `name` (`str`) – 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` (`int`) – Size of the variable. The variable becomes a one-dimensional vector of the given size.
- `ldom` (*LinearDomain*) – A linear domain for the variable.
- `rdom` (*RangeDomain*) – A ranged domain for the variable.
- `qdom` (*ConeDomain*) – A conic domain for the variable.
- `shp` (`int`[]) – Defines the shape of the variable.
- `psddom` (*PSDDomain*) – A semidefinte domain for the variable.
- `n` (`int`) – Dimension of the semidefinite variable.
- `m` (`int`) – Number of semidefinite variables.
- `lpsddom` (*LinPSDDomain*) – A linear semidefinte domain for the variable.

Return (*Variable*)

`Model.writeTask`

```
writeTask(str filename)
```

Dump the current solver task to a file. The file extension determines the file format, see Section *Supported File Formats* for details. The file can be read with the command line **MOSEK** or with the Optimizer API for debugging purposes.

Parameters `filename` (`str`) – Name of the output file.

14.2.19 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*.

Implements *Constraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.
Constraint.getModel – Return the model that the constraint belongs to.
Constraint.getND – Return the number of dimensions in the constraint shape.
Constraint.getShape – Return the constraint's shape.
Constraint.getSize – Return the total number of elements in the constraint.
Constraint.index – Get a single element from a constraint.
Constraint.level – Get the primal solution values of the constraint.
Constraint.slice – Create a slice constraint.
Constraint.update – Update part of a constraint.
ModelConstraint.toString – Create a human readable string representation of the constraint.

Implemented by *ConicConstraint*, *PSDConstraint*, *RangedConstraint*, *LinearPSDConstraint*, *LinearConstraint*

ModelConstraint.toString

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

14.2.20 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*.

Implements *BaseVariable*

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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this 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 *RangedVariable*, *ConicVariable*, *LinearVariable*, *LinearPSDVariable*, *PSDVariable*, *SymLinearVariable*, *SymRangedVariable*

14.2.21 Class NDSparseArray

`mosek.fusion.NDSparseArray`

Representation of a sparse n-dimensional array.

Static members *NDSparseArray.make* – Create a sparse n-dimensional matrix (tensor).

`NDSparseArray.make`

```
NDSparseArray.make(int[] dims, int[][] sub, float[] cof) -> NDSparseArray
NDSparseArray.make(int[] dims, int[] inst, float[] cof) -> NDSparseArray
NDSparseArray.make(Matrix m) -> NDSparseArray
```

Create a sparse n-dimensional matrix (tensor).

Parameters

- `dims` (`int`[]) – Dimensions.
- `sub` (`int`[][]) – Positions of nonzeros. Array where each row is an n-dimensional index.
- `cof` (`float`[]) – Values of nonzero elements. Array of coefficients corresponding to subscripts.
- `inst` (`int`[]) – Positions of nonzeros using linear indexes into the array.
- `m` (*Matrix*) – An initializing matrix.

Return (*NDSparseArray*)

14.2.22 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

Implements *ModelConstraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.
Constraint.getModel – Return the model that the constraint belongs to.
Constraint.getND – Return the number of dimensions in the constraint shape.
Constraint.getShape – Return the constraint's shape.
Constraint.getSize – Return the total number of elements in the constraint.
Constraint.index – Get a single element from a constraint.
Constraint.level – Get the primal solution values of the constraint.
Constraint.slice – Create a slice constraint.
Constraint.update – Update part of a constraint.
PSDConstraint.toString – Create a human readable string representation of the constraint.

`PSDConstraint.toString`

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

14.2.23 Class PSDDomain

`mosek.fusion.PSDDomain`

Represent the domain od PSD matrices.

Members *PSDDomain.axis* – Set the dimension along which the cones are created.

`PSDDomain.axis`

```
axis(int conedim1, int conedim2) -> PSDDomain
```

Set the dimension along which the cones are created.

Parameters

- `conedim1` (`int`)
- `conedim2` (`int`)

Return (*PSDDomain*)

14.2.24 Class PSDVariable

`mosek.fusion.PSDVariable`

This class represents a positive semidefinite variable.

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.

BaseVariable.getSize – Get the total number of elements in the variable.

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. Has no effect on elements of semidefinite matrix variables.

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.

BaseVariable.setLevel – Input solution values for this variable

BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension

BaseVariable.transpose – Transpose a vector or matrix variable

PSDVariable.toString – Create a string representation of the variable.

`PSDVariable.toString`

```
toString() -> str
```

Create a string representation of the variable.

Return (`str`)

14.2.25 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.

RangeDomain.withShape – Set the shape of the domain.

Implemented by *SymmetricRangeDomain*

RangeDomain.integral

```
integral() -> RangeDomain
```

Modify a given domain restricting its elements to be integral.

Return (*RangeDomain*)

RangeDomain.sparse

```
sparse() -> RangeDomain
sparse(int[] sparsity) -> RangeDomain
sparse(int[][] sparsity) -> RangeDomain
```

Creates a domain exploiting sparsity.

Parameters

- sparsity (int[])
- sparsity (int[][])

Return (*RangeDomain*)

RangeDomain.symmetric

```
symmetric() -> SymmetricRangeDomain
```

Creates a symmetric domain.

Return (*SymmetricRangeDomain*)

RangeDomain.withShape

```
withShape(int[] shp) -> RangeDomain
```

Set the shape of the domain.

Parameters shp (int[]) – The shape of the domain

Return (*RangeDomain*)

14.2.26 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).

Implements *ModelConstraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.slice – Create a slice constraint.

Constraint.update – Update part of a constraint.

ModelConstraint.toString – Create a human readable string representation of the constraint.

14.2.27 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).

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.

BaseVariable.getSize – Get the total number of elements in the variable.

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. Has no effect on elements of semidefinite matrix variables.

BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.

BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.

BaseVariable.setLevel – Input solution values for this 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

14.2.28 Class Set

`mosek.fusion.Set`

This class contains static methods for creating and manipulating shape specifications.

Static members *Set.make* – Creates a shape.

Set.scalar – Create a shape of size 1.

Set.strides – Compute the strides from a shape.

Set.make

```
Set.make(str[] names) -> int[]
Set.make(int sz) -> int[]
Set.make(int s1, int s2) -> int[]
Set.make(int s1, int s2, int s3) -> int[]
Set.make(int[] sizes) -> int[]
Set.make(int[] set1, int[] set2) -> int[]
```

This static method is a factory for different kind of set objects:

- A (multi-dimensional) set of integers (shape).
- A set whose elements are strings.
- A set obtained as Cartesian product of sets given in a list.

Parameters
- `names` (`str`[]) – A list of strings for a set of strings.
- `sz` (`int`) – The size of a one-dimensional set of integers.
- `s1` (`int`) – Size of the first dimension.
- `s2` (`int`) – Size of the second dimension.
- `s3` (`int`) – Size of the third dimension.
- `sizes` (`int`[]) – The sizes of dimensions for a multi-dimensional integer set.
- `set1` (`int`[]) – First factor in a Cartesian product.
- `set2` (`int`[]) – Second factor in a Cartesian product.

Return (`int`[])

Set.scalar

```
Set.scalar() -> int[]
```

Create a shape of size 1.

Return (`int`[])

Set.strides

```
Set.strides(int[] shape) -> int[]
```

Compute the strides from a shape.

Parameters `shape` (`int`[])
Return (`int`[])

14.2.29 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.

Implements *Constraint*

Members *Constraint.dual* – Get the dual solution values of the constraint.
Constraint.getModel – Return the model that the constraint belongs to.
Constraint.getND – Return the number of dimensions in the constraint shape.
Constraint.getShape – Return the constraint's shape.
Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.
Constraint.level – Get the primal solution values of the constraint.
Constraint.slice – Create a slice constraint.
Constraint.update – Update part of a constraint.
SliceConstraint.toString – Create a human readable string representation of the constraint.

`SliceConstraint.toString`

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (`str`)

14.2.30 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.

Implements *BaseVariable*

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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this 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

14.2.31 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.

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this 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

14.2.32 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).

Implements *ModelVariable*

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.eval – Evaluate the expression and push the result onto the work stack.
BaseVariable.getDim – Return the d'th dimension in the expression.
BaseVariable.getModel – Get the *Model* object that the variable belongs to.
BaseVariable.getND – Get the number of dimensions in the variable shape.
BaseVariable.getShape – Get the variable shape.
BaseVariable.getSize – Get the total number of elements in the variable.
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. Has no effect on elements of semidefinite matrix variables.
BaseVariable.pick – Create a slice variable by picking a list of indexes from this variable.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this 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

14.2.33 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 string representation of the expression.

`SymmetricExpr.toString`

```
toString() -> str
```

Returns a human readable string representation of the expression.

Return (`str`)

14.2.34 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.

`SymmetricLinearDomain.integral`

```
integral() -> SymmetricLinearDomain
```

Modify a given domain restricting its elements to be integral.

Return (*SymmetricLinearDomain*)

`SymmetricLinearDomain.sparse`

```
sparse(int[] sparsity) -> SymmetricLinearDomain
sparse(int[][] sparsity) -> SymmetricLinearDomain
```

Modify a given domain exploting sparsity, i.e only instantiating the variables that are actually used in the model.

Parameters

- `sparsity` (int[])
- `sparsity` (int[][])

Return (*SymmetricLinearDomain*)

14.2.35 Class SymmetricRangeDomain

`mosek.fusion.SymmetricRangeDomain`

Represent a ranged domain with symmetry.

Implements *RangeDomain*

Members *RangeDomain.integral* – Creates a domain of integral variables.
RangeDomain.sparse – Creates a domain exploiting sparsity.
RangeDomain.symmetric – Creates a symmetric domain.
RangeDomain.withShape – Set the shape of the domain.

14.2.36 Class SymmetricVariable

`mosek.fusion.SymmetricVariable`

In some cases using this variable instead of a standard *Variable* can reduce the number of instantiated variables by approximately *50%*.

Implements *Variable*

Members *Expression.eval* – Evaluate the expression and push the result onto the work stack.
Expression.getDim – Return the d'th dimension in the expression.
Variable.antidiag – Return the anti-diagonal of a variable matrix.
Variable.asExpr – Create an expression corresponding to the variable object.
Variable.diag – Return the diagonal of a variable matrix.
Variable.dual – Get the dual solution value of the variable.
Variable.getModel – Return the model to which the variable belongs.
Variable.getND – Get the number of dimensions in the variable shape.
Variable.getShape – Return the shape of the variable.
Variable.getSize – Get the total number of elements in the variable.
Variable.index – Return a single entry in the variable.
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. Has no effect on elements of semidefinite matrix variables.
Variable.pick – Create a one-dimensional variable by picking a list of indexes from this variable.
Variable.reshape – Reshape the variable. The new shape must have the same total size as the current.
Variable.setLevel – Input solution values for this 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 the variable.

Implemented by *PSDVariable*, *SymLinearVariable*, *SymRangedVariable*

14.2.37 Class Var

`mosek.fusion.Var`

Contains several static methods for manipulating variable objects and creating new variables from old ones.

Primal and dual solution values and additional operations on variables are available from the *Variable* class.

Static members *Var.compress* – Reshape a variable object by removing all dimensions of size 1.
Var.empty – Produce a new empty variable of the given shape.
Var.flatten – Create a one-dimensional logical view of a variable object.
Var.hrepeat – Repeat a variable a number of times in the second dimension.
Var.hstack – Stack a list of variables horizontally (i.e. along the second dimension).
Var.promote – Pad variable shape.
Var.repeat – Repeat a variable a number of times in the given dimension.
Var.reshape – Create a reshaped view of the given variable.
Var.stack – Stack a list of variables in an arbitrary dimension.
Var.vrepeat – Repeat a variable a number of times in the first dimension.
Var.vstack – Stack a list of variables vertically (i.e. along the first dimension).

`Var.compress`

```
Var.compress(Variable v) -> Variable
```

Reshape a variable object by removing all dimensions of size 1. The result contains the same number of elements, but all dimensions are larger than 1 (except if the original variable contains exactly one element).

Parameters `v` (*Variable*) – The variable object to compress.
Return (*Variable*)

`Var.empty`

```
Var.empty(int[] shape) -> Variable
```

Produce a new empty variable of the given shape.

Parameters `shape` (`int`[]) – The shape of a variable.
Return (*Variable*)

`Var.flatten`

```
Var.flatten(Variable v) -> Variable
```

Create a one-dimensional *Variable* object that represents a *logical* view of the $k-$dimensional variable V. The V matrix is traversed starting from the innermost dimension. For a two-dimensional matrix this means it is traversed row after row.

The returned view is a one-dimensional array of size equal to the product of dimensions of V.

Parameters `v` (*Variable*) – The variable to be flattened.
Return (*Variable*)

`Var.hrepeat`

```
Var.hrepeat(Variable v, int n) -> Variable
```

Repeat a variable a number of times in the second dimension. This is equivalent to horizontal stacking of n copies of the variable; see *Var.hstack*.

Parameters
- `v` (*Variable*) – A variable object.
- `n` (`int`) – Number of times to repeat `v`.

Return (*Variable*)

Var.hstack

```
Var.hstack(Variable[] v) -> Variable
Var.hstack(Variable v1, Variable v2) -> Variable
Var.hstack(Variable v1, Variable v2, Variable v3) -> Variable
```

Stack a list of variables horizontally (i.e. along the second dimension). The variables must have the same shape, except for the second dimension.

For example, if x^1, x^2, x^3 are three one-dimensional variables of length n then their horizontal stack is the matrix variable

$$\begin{bmatrix} | & | & | \\ x^1 & x^2 & x^3 \\ | & | & | \end{bmatrix}$$

of shape `(n,3)`.

Parameters

- `v` (*Variable*[]) – List of variables to stack.
- `v1` (*Variable*) – First variable in the stack.
- `v2` (*Variable*) – Second variable in the stack.
- `v3` (*Variable*) – Third variable in the stack.

Return (*Variable*)

Var.promote

```
Var.promote(Variable v, int nd) -> Variable
```

Pad the variable shape up to `nd` dimensions.

Parameters

- `v` (*Variable*) – Variable to pad.
- `nd` (`int`) – Final number of dimensions.

Return (*Variable*)

Var.repeat

```
Var.repeat(Variable v, int dim, int n) -> Variable
Var.repeat(Variable v, int n) -> Variable
```

Repeat a variable a number of times in the given dimension. If `dim` is non-negative this is equivalent to stacking n copies of the variable in dimension `dim`; see *Var.stack*.

If `dim` is negative then a new dimension is added in front, so the new variable has shape `(n, v.shape())`.

The default is repeating in the first dimension as in *Var.vrepeat*.

Parameters

- `v` (*Variable*) – A variable object.
- `dim` (`int`) – Dimension to repeat in.
- `n` (`int`) – Number of times to repeat `v`.

Return (*Variable*)

Var.reshape

```
Var.reshape(Variable v, int[] shape) -> Variable
Var.reshape(Variable v, int d1, int d2) -> Variable
Var.reshape(Variable v, int d1) -> Variable
```

Create a reshaped view of the given variable.

Parameters

- `v` (*Variable*) – A variable object.
- `shape` (`int`[]) – An array containing the shape of the new variable.
- `d1` (`int`) – Size of first dimension in the result.
- `d2` (`int`) – Size of second dimension in the result.

Return (*Variable*)

`Var.stack`

```
Var.stack(Variable[] v, int dim) -> Variable
Var.stack(Variable v1, Variable v2, int dim) -> Variable
Var.stack(Variable v1, Variable v2, Variable v3, int dim) -> Variable
Var.stack(Variable[][] vlist) -> Variable
```

Stack a list of variables along an arbitrary dimension. All variables must have the same shape, except for dimension `dim`.

For example, suppose x, y are two matrix variables of shape $n \times m$. Then stacking them in the first dimension produces a matrix variable of shape `(2n,m)`:

$$\begin{bmatrix} x \\ y \end{bmatrix},$$

stacking them in the second dimension produces a matrix variable of shape `(n,2m)`:

$$\begin{bmatrix} x & y \end{bmatrix},$$

and stacking them in the third dimension produces a three-dimensional variable of shape `(n,m,2)`.

The version which takes a two-dimensional array of variables constructs a block matrix variable with the given variables as blocks. The dimensions of the blocks must be suitably compatible. The variables may be more than two-dimensional, if they have the same size in the remaining dimensions; the block stacking still takes place in the first and second dimension.

Parameters

- `v` (*Variable*[]) – List of variables to stack.
- `dim` (`int`) – 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*[][]) – List of variables to stack.

Return (*Variable*)

`Var.vrepeat`

```
Var.vrepeat(Variable v, int n) -> Variable
```

Repeat a variable a number of times in the first dimension. This is equivalent to vertically stacking of n copies of the variable; see *Var.vstack*.

Parameters

- `v` (*Variable*) – A variable object.
- `n` (`int`) – Number of times to repeat `v`.

Return (*Variable*)

`Var.vstack`

```
Var.vstack(Variable[] v) -> Variable
Var.vstack(Variable v1, Variable v2) -> Variable
Var.vstack(Variable v1, Variable v2, Variable v3) -> Variable
```

Stack a list of variables vertically (i.e. along the first dimension). The variables must have the same shape, except for the first dimension.

For example, if y^1, y^2, y^3 are three horizontal vector variables of length n (and shape `(1,n)`) then their vertical stack is the matrix variable

$$\begin{bmatrix} -y^1- \\ -y^2- \\ -y^3- \end{bmatrix}$$

of shape `(3,n)`.

Parameters

- `v` (*Variable*[]) – List of variables to stack.
- `v1` (*Variable*) – First variable in the stack.
- `v2` (*Variable*) – Second variable in the stack.
- `v3` (*Variable*) – Third variable in the stack.

Return (*Variable*)

14.2.38 Class Variable

`mosek.fusion.Variable`

An abstract variable object. This is the base class for all variable types in *Fusion*, and it contains several 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.

More static methods to manipulate variables are available in the *Var* class.

Implements *Expression*

Members *Expression.eval* – Evaluate the expression and push the result onto the work stack.

Expression.getDim – Return the d'th dimension in the expression.

Variable.antidiag – Return the anti-diagonal of a variable matrix.

Variable.asExpr – Create an expression corresponding to the variable object.

Variable.diag – Return the diagonal of a variable matrix.

Variable.dual – Get the dual solution value of the variable.

Variable.getModel – Return the model to which the variable belongs.

Variable.getND – Get the number of dimensions in the variable shape.

Variable.getShape – Return the shape of the variable.

Variable.getSize – Get the total number of elements in the variable.

Variable.index – Return a single entry in the variable.

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. Has no effect on elements of semidefinite matrix variables.

Variable.pick – Create a one-dimensional variable by picking a list of indexes from this variable.

Variable.reshape – Reshape the variable. The new shape must have the same total size as the current.

Variable.setLevel – Input solution values for this 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 the variable.

Implemented by *BaseVariable*, *SymmetricVariable*

Variable.antidiag

```
antidiag(int index) -> Variable
antidiag() -> Variable
```

Return the anti-diagonal of a variable matrix in a one-dimensional variable object. The main anti-diagonal is defined as starting with the element in the first row and last column.

Parameters index (int) – Index of the anti-diagonal. Index 0 means the main anti-diagonal, negative indices are below it and positive indices are above it.

Return (*Variable*)

Variable.asExpr

```
asExpr() -> Expression
```

Create an *Expression* object corresponding to $I \cdot V$, where I is the identity matrix and V is this variable.

Return (*Expression*)

Variable.diag

```
diag(int index) -> Variable
diag() -> Variable
```

Return the diagonal of a square variable matrix in a one-dimensional variable object. The main diagonal is defined as starting with the element in the first row and first column.

Parameters index (int) – Index of the diagonal. Index 0 means the main diagonal, negative indices are below it and positive indices are above it.

Return (*Variable*)

Variable.dual

```
dual() -> float[]
```

Get the dual solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (float[])

Variable.getModel

```
getModel() -> Model
```

Get the *Model* object that the variable belongs to.

Return (*Model*)

Variable.getND

```
getND() -> int
```

Get the number of dimensions in the variable shape.

Return (int)

`Variable.getShape`

```
getShape() -> int[]
```

Get the variable shape.

Return (int[])

`Variable.getSize`

```
getSize() -> int
```

Get the total number of elements in the variable.

Return (int)

`Variable.index`

```
index(int i1) -> Variable
index(int i1, int i2) -> Variable
index(int i1, int i2, int i3) -> Variable
index(int[] idx) -> Variable
```

Return a variable of size one corresponding to a single element in the variable object.

Parameters

- `i1` (int) – Index in the first dimension of the element requested.
- `i2` (int) – Index in the second dimension of the element requested.
- `i3` (int) – Index in the third dimension of the element requested.
- `idx` (int[]) – List of indexes of the elements requested.

Return (*Variable*)

`Variable.level`

```
level() -> float[]
```

Get the primal solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (float[])

`Variable.makeContinuous`

```
makeContinuous()
```

Drop integrality constraints on the variable, if any.

`Variable.makeInteger`

```
makeInteger()
```

Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.

`Variable.pick`

```
pick(int[] idxs) -> Variable
pick(int[][] midxs) -> Variable
pick(int[] i1, int[] i2) -> Variable
pick(int[] i1, int[] i2, int[] i3) -> Variable
```

Create a one-dimensional variable by picking a list of indexes from this variable.

Parameters

- `idxs` (`int`[]) – Indexes of the elements requested.
- `midxs` (`int`[][]) – A sequence of multi-dimensional indexes of the elements requested.
- `i1` (`int`[]) – Index along the first dimension.
- `i2` (`int`[]) – Index along the second dimension.
- `i3` (`int`[]) – Index along the third dimension.

Return (*Variable*)

`Variable.reshape`

```
reshape(int[] shape) -> Variable
reshape(int dim0) -> Variable
reshape(int dim0, int dim1) -> Variable
reshape(int dim0, int dim1, int dim2) -> Variable
```

Reshape the variable. The new shape must have the same total size as the current.

Parameters

- `shape` (`int`[]) – The new shape.
- `dim0` (`int`) – First dimension of new shape
- `dim1` (`int`) – Second dimension of new shape
- `dim2` (`int`) – Third dimension of new shape

Return (*Variable*)

`Variable.setLevel`

```
setLevel(float[] v)
```

Set values for an initial solution for this variable. Note that these values are buffered until the solver is called; they are not available through the `level()` methods.

Parameters `v` (`float`[]) – An array of values to be assigned to the variable.

`Variable.slice`

```
slice(int first, int last) -> Variable
slice(int[] firsta, int[] lasta) -> Variable
```

Create a slice variable by picking a range of indexes for each variable dimension.

Parameters

- `first` (`int`) – The index from which the slice begins.
- `last` (`int`) – The index after the last element of the slice.
- `firsta` (`int`[]) – The indices from which the slice of a multidimensional variable begins.
- `lasta` (`int`[]) – The indices after the last element of slice of a multidimensional variable.

Return (*Variable*)

Variable.toString

```
toString() -> str
```

Create a string representation of the variable.

Return (str)

Variable.transpose

```
transpose() -> Variable
```

Return the transpose of the current variable. The variable must have at most two dimensions.

Return (*Variable*)

14.2.39 Class WorkStack

mosek.fusion.WorkStack

Stack object used to store expression evaluations. For internal use.

Members *WorkStack.allocf64* – Allocate n doubles and return the stack index of the first

WorkStack.alloci32 – Allocate n int32s and return the stack index of the first

WorkStack.alloci64 – Allocate n int64s and return the stack index of the first

WorkStack.clear – Clear all stacks

WorkStack.ensuref64 – Make sure that the double stack has capacity for n new items

WorkStack.ensurei32 – Make sure that the int32 stack has capacity for n new items

WorkStack.ensurei64 – Make sure that the int64 stack has capacity for n new items

WorkStack.peekf64 – Peek at one double item from stack

WorkStack.peeki32 – Peek at one int32 item from stack

WorkStack.peeki64 – Peek at one int64 item from stack

WorkStack.popf64 – Pop one or more double items from stack

WorkStack.popi32 – Pop one or more int32 items from stack

WorkStack.popi64 – Pop one or more int64 items from stack

WorkStack.pushf64 – Push an double value

WorkStack.pushi32 – Push an int32 value

WorkStack.pushi64 – Push an int64 value

WorkStack.allocf64

```
allocf64(int n) -> int
```

Allocate n doubles and return the stack index of the first

Parameters n (int)

Return (int)

WorkStack.alloci32

```
alloci32(int n) -> int
```

Allocate n int32s and return the stack index of the first

Parameters n (int)

Return (int)

`WorkStack.alloci64`

```
alloci64(int n) -> int
```

Allocate n int64s and return the stack index of the first

Parameters `n` (`int`)
Return (`int`)

`WorkStack.clear`

```
clear()
```

Clear all stacks

`WorkStack.ensuref64`

```
ensuref64(int n)
```

Make sure that the double stack has capacity for n new items

Parameters `n` (`int`) – Number of items to make space for

`WorkStack.ensurei32`

```
ensurei32(int n)
```

Make sure that the int32 stack has capacity for n new items

Parameters `n` (`int`) – Number of items to make space for

`WorkStack.ensurei64`

```
ensurei64(int n)
```

Make sure that the int64 stack has capacity for n new items

Parameters `n` (`int`) – Number of items to make space for

`WorkStack.peekf64`

```
peekf64(int i) -> float
peekf64() -> float
```

Peek at one double item from stack

Parameters `i` (`int`) – Peek at this index on the stack (default is 0)
Return (`float`)

`WorkStack.peeki32`

```
peeki32(int i) -> int
peeki32() -> int
```

Peek at one int32 item from stack

Parameters `i` (`int`) – Peek at this index on the stack (default is 0)
Return (`int`)

WorkStack.peeki64

```
peeki64(int i) -> int
peeki64() -> int
```

Peek at one int64 item from stack

Parameters i (int) – Peek at this index on the stack (default is 0)

Return (int)

WorkStack.popf64

```
popf64() -> float
popf64(int n, float[] r, int ofs)
popf64(int n) -> int
```

Pop one or more double items from stack

Parameters
- n (int) – Number of items to pop (default is 1)
- r (float[]) – Copy popped item to this array
- ofs (int) – Copy popped items to this offset in *r*

Return
- (float)
- (int)

WorkStack.popi32

```
popi32() -> int
popi32(int n, int[] r, int ofs)
popi32(int n) -> int
```

Pop one or more int32 items from stack

Parameters
- n (int) – Number of items to pop (default is 1)
- r (int[]) – Copy popped item to this array
- ofs (int) – Copy popped items to this offset in *r*

Return (int)

WorkStack.popi64

```
popi64() -> int
popi64(int n, int[] r, int ofs)
popi64(int n) -> int
```

Pop one or more int64 items from stack

Parameters
- n (int) – Number of items to pop (default is 1)
- r (int[]) – Copy popped item to this array
- ofs (int) – Copy popped items to this offset in *r*

Return (int)

WorkStack.pushf64

```
pushf64(float v)
```

Push an double value

Parameters `v` (`float`) – The value to store

`WorkStack.pushi32`

```
pushi32(int v)
```

Push an int32 value

Parameters `v` (`int`) – The value to store

`WorkStack.pushi64`

```
pushi64(int v)
```

Push an int64 value

Parameters `v` (`int`) – The value to store

14.3 Parameters grouped by topic

Basis identification

- *simLuTolRelPiv*
- *biCleanOptimizer*
- *biIgnoreMaxIter*
- *biIgnoreNumError*
- *biMaxIterations*
- *intpntBasis*
- *logBi*
- *logBiFreq*

Conic interior-point method

- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*
- *intpntCoTolPfeas*
- *intpntCoTolRelGap*

Data input/output

- *logFile*
- *writeLpFullObj*
- *writeLpLineWidth*
- *writeLpQuotedNames*
- *writeLpTermsPerLine*
- *basSolFileName*
- *dataFileName*
- *intSolFileName*
- *itrSolFileName*
- *writeLpGenVarName*

Dual simplex

- *simDualCrash*
- *simDualRestrictSelection*
- *simDualSelection*

Infeasibility report

- *logInfeasAna*

Interior-point method

- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*
- *intpntCoTolPfeas*
- *intpntCoTolRelGap*
- *intpntTolDfeas*
- *intpntTolDsafe*
- *intpntTolInfeas*
- *intpntTolMuRed*
- *intpntTolPath*
- *intpntTolPfeas*
- *intpntTolPsafe*
- *intpntTolRelGap*
- *intpntTolRelStep*

- *intpntTolStepSize*
- *biIgnoreMaxIter*
- *biIgnoreNumError*
- *intpntBasis*
- *intpntDiffStep*
- *intpntMaxIterations*
- *intpntMaxNumCor*
- *intpntOffColTrh*
- *intpntOrderGpNumSeeds*
- *intpntOrderMethod*
- *intpntRegularizationUse*
- *intpntScaling*
- *intpntSolveForm*
- *intpntStartingPoint*
- *logIntpnt*

License manager

- *cacheLicense*
- *licenseDebug*
- *licensePauseTime*
- *licenseSuppressExpireWrns*
- *licenseTrhExpiryWrn*
- *licenseWait*

Logging

- *log*
- *logBi*
- *logBiFreq*
- *logCutSecondOpt*
- *logExpand*
- *logFile*
- *logInfeasAna*
- *logIntpnt*
- *logLocalInfo*
- *logMio*
- *logMioFreq*

- *logOrder*
- *logPresolve*
- *logResponse*
- *logSim*
- *logSimFreq*

Mixed-integer optimization

- *mioMaxTime*
- *mioRelGapConst*
- *mioTolAbsGap*
- *mioTolAbsRelaxInt*
- *mioTolFeas*
- *mioTolRelDualBoundImprovement*
- *mioTolRelGap*
- *logMio*
- *logMioFreq*
- *mioBranchDir*
- *mioConicOuterApproximation*
- *mioCutClique*
- *mioCutCmir*
- *mioCutGmi*
- *mioCutImpliedBound*
- *mioCutKnapsackCover*
- *mioCutSelectionLevel*
- *mioFeaspumpLevel*
- *mioHeuristicLevel*
- *mioMaxNumBranches*
- *mioMaxNumRelaxs*
- *mioMaxNumRootCutRounds*
- *mioMaxNumSolutions*
- *mioNodeOptimizer*
- *mioNodeSelection*
- *mioPerspectiveReformulate*
- *mioProbingLevel*
- *mioPropagateObjectiveConstraint*
- *mioRinsMaxNodes*

- *mioRootOptimizer*
- *mioRootRepeatPresolveLevel*
- *mioSeed*
- *mioVbDetectionLevel*

Output information

- *licenseSuppressExpireWrns*
- *licenseTrhExpiryWrn*
- *log*
- *logBi*
- *logBiFreq*
- *logCutSecondOpt*
- *logExpand*
- *logFile*
- *logInfeasAna*
- *logIntpnt*
- *logLocalInfo*
- *logMio*
- *logMioFreq*
- *logOrder*
- *logResponse*
- *logSim*
- *logSimFreq*
- *logSimMinor*

Overall solver

- *biCleanOptimizer*
- *infeasPreferPrimal*
- *licenseWait*
- *mioMode*
- *optimizer*
- *presolveLevel*
- *presolveUse*

Overall system

- *autoUpdateSolInfo*
- *intpntMultiThread*
- *licenseWait*
- *mtSpincount*
- *numThreads*
- *removeUnusedSolutions*
- *remoteAccessToken*

Presolve

- *presolveTolAbsLindep*
- *presolveTolAij*
- *presolveTolRelLindep*
- *presolveTolS*
- *presolveTolX*
- *presolveEliminatorMaxFill*
- *presolveEliminatorMaxNumTries*
- *presolveLevel*
- *presolveLindepAbsWorkTrh*
- *presolveLindepRelWorkTrh*
- *presolveLindepUse*
- *presolveMaxNumPass*
- *presolveUse*

Primal simplex

- *simPrimalCrash*
- *simPrimalRestrictSelection*
- *simPrimalSelection*

Simplex optimizer

- *basisRelTolS*
- *basisTolS*
- *basisTolX*
- *simLuTolRelPiv*
- *simplexAbsTolPiv*
- *logSim*

- *logSimFreq*
- *logSimMinor*
- *simBasisFactorUse*
- *simDegen*
- *simDualPhaseoneMethod*
- *simExploitDupvec*
- *simHotstart*
- *simHotstartLu*
- *simMaxIterations*
- *simMaxNumSetbacks*
- *simNonSingular*
- *simPrimalPhaseoneMethod*
- *simRefactorFreq*
- *simReformulation*
- *simSaveLu*
- *simScaling*
- *simScalingMethod*
- *simSeed*
- *simSolveForm*
- *simSwitchOptimizer*

Solution input/output

- *basSolFileName*
- *intSolFileName*
- *itrSolFileName*

Termination criteria

- *basisRelTolS*
- *basisTolS*
- *basisTolX*
- *intpntCoTolDfeas*
- *intpntCoTolInfeas*
- *intpntCoTolMuRed*
- *intpntCoTolNearRel*
- *intpntCoTolPfeas*
- *intpntCoTolRelGap*

- *intpntTolDfeas*
- *intpntTolInfeas*
- *intpntTolMuRed*
- *intpntTolPfeas*
- *intpntTolRelGap*
- *lowerObjCut*
- *lowerObjCutFiniteTrh*
- *mioMaxTime*
- *mioRelGapConst*
- *mioTolRelGap*
- *optimizerMaxTime*
- *upperObjCut*
- *upperObjCutFiniteTrh*
- *biMaxIterations*
- *intpntMaxIterations*
- *mioMaxNumBranches*
- *mioMaxNumRootCutRounds*
- *mioMaxNumSolutions*
- *simMaxIterations*

14.4 Parameters (alphabetical list sorted by type)

- *Double parameters*
- *Integer parameters*
- *String parameters*

14.4.1 Double parameters

`"basisRelTolS"`

Maximum relative dual bound violation allowed in an optimal basic solution.

Default 1.0e-12
Accepted [0.0; +inf]
Example `M.setSolverParam("basisRelTolS", 1.0e-12)`
Generic name MSK_DPAR_BASIS_REL_TOL_S
Groups *Simplex optimizer*, *Termination criteria*

`"basisTolS"`

Maximum absolute dual bound violation in an optimal basic solution.

Default 1.0e-6
Accepted [1.0e-9; +inf]
Example `M.setSolverParam("basisTolS", 1.0e-6)`
Generic name MSK_DPAR_BASIS_TOL_S

Groups *Simplex optimizer*, *Termination criteria*

"basisTolX"

Maximum absolute primal bound violation allowed in an optimal basic solution.

Default 1.0e-6

Accepted [1.0e-9; +inf]

Example `M.setSolverParam("basisTolX", 1.0e-6)`

Generic name MSK_DPAR_BASIS_TOL_X

Groups *Simplex optimizer*, *Termination criteria*

"intpntCoTolDfeas"

Dual feasibility tolerance used by the interior-point optimizer for conic problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntCoTolDfeas", 1.0e-8)`

See also *intpntCoTolNearRel*

Generic name MSK_DPAR_INTPNT_CO_TOL_DFEAS

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

"intpntCoTolInfeas"

Infeasibility tolerance used by the interior-point optimizer for conic problems. Controls when the interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Default 1.0e-12

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntCoTolInfeas", 1.0e-12)`

Generic name MSK_DPAR_INTPNT_CO_TOL_INFEAS

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

"intpntCoTolMuRed"

Relative complementarity gap tolerance used by the interior-point optimizer for conic problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntCoTolMuRed", 1.0e-8)`

Generic name MSK_DPAR_INTPNT_CO_TOL_MU_RED

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

"intpntCoTolNearRel"

Optimality tolerance used by the interior-point optimizer for conic problems. If **MOSEK** cannot compute a solution that has the prescribed accuracy then it will check if the solution found satisfies the termination criteria with all tolerances multiplied by the value of this parameter. If yes, then the solution is also declared optimal.

Default 1000

Accepted [1.0; +inf]

Example `M.setSolverParam("intpntCoTolNearRel", 1000)`

Generic name MSK_DPAR_INTPNT_CO_TOL_NEAR_REL

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

"intpntCoTolPfeas"

Primal feasibility tolerance used by the interior-point optimizer for conic problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntCoTolPfeas", 1.0e-8)`

See also *intpntCoTolNearRel*

Generic name MSK_DPAR_INTPNT_CO_TOL_PFEAS

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

`"intpntCoTolRelGap"`

Relative gap termination tolerance used by the interior-point optimizer for conic problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntCoTolRelGap", 1.0e-8)`

See also *intpntCoTolNearRel*

Generic name MSK_DPAR_INTPNT_CO_TOL_REL_GAP

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

`"intpntTolDfeas"`

Dual feasibility tolerance used by the interior-point optimizer for linear problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntTolDfeas", 1.0e-8)`

Generic name MSK_DPAR_INTPNT_TOL_DFEAS

Groups *Interior-point method*, *Termination criteria*

`"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.

Default 1.0

Accepted [1.0e-4; +inf]

Example `M.setSolverParam("intpntTolDsafe", 1.0)`

Generic name MSK_DPAR_INTPNT_TOL_DSAFE

Groups *Interior-point method*

`"intpntTolInfeas"`

Infeasibility tolerance used by the interior-point optimizer for linear problems. Controls when the interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Default 1.0e-10

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntTolInfeas", 1.0e-10)`

Generic name MSK_DPAR_INTPNT_TOL_INFEAS

Groups *Interior-point method*, *Termination criteria*

`"intpntTolMuRed"`

Relative complementarity gap tolerance used by the interior-point optimizer for linear problems.

Default 1.0e-16

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntTolMuRed", 1.0e-16)`

Generic name MSK_DPAR_INTPNT_TOL_MU_RED

Groups *Interior-point method*, *Termination criteria*

`"intpntTolPath"`

Controls how close the interior-point optimizer follows the central path. A large value of this parameter means the central path is followed very closely. On numerically unstable problems it may be worthwhile to increase this parameter.

Default 1.0e-8

Accepted [0.0; 0.9999]

Example `M.setSolverParam("intpntTolPath", 1.0e-8)`

Generic name MSK_DPAR_INTPNT_TOL_PATH

Groups *Interior-point method*

`"intpntTolPfeas"`

Primal feasibility tolerance used by the interior-point optimizer for linear problems.

Default 1.0e-8

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntTolPfeas", 1.0e-8)`

Generic name MSK_DPAR_INTPNT_TOL_PFEAS

Groups *Interior-point method*, *Termination criteria*

`"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.

Default 1.0

Accepted [1.0e-4; +inf]

Example `M.setSolverParam("intpntTolPsafe", 1.0)`

Generic name MSK_DPAR_INTPNT_TOL_PSAFE

Groups *Interior-point method*

`"intpntTolRelGap"`

Relative gap termination tolerance used by the interior-point optimizer for linear problems.

Default 1.0e-8

Accepted [1.0e-14; +inf]

Example `M.setSolverParam("intpntTolRelGap", 1.0e-8)`

Generic name MSK_DPAR_INTPNT_TOL_REL_GAP

Groups *Termination criteria*, *Interior-point method*

`"intpntTolRelStep"`

Relative step size to the boundary for linear and quadratic optimization problems.

Default 0.9999

Accepted [1.0e-4; 0.999999]

Example `M.setSolverParam("intpntTolRelStep", 0.9999)`

Generic name MSK_DPAR_INTPNT_TOL_REL_STEP

Groups *Interior-point method*

`"intpntTolStepSize"`

Minimal step size tolerance. 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 to stop.

Default 1.0e-6

Accepted [0.0; 1.0]

Example `M.setSolverParam("intpntTolStepSize", 1.0e-6)`

Generic name MSK_DPAR_INTPNT_TOL_STEP_SIZE

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.

Default -1.0e30

Accepted [-inf; +inf]

Example `M.setSolverParam("lowerObjCut", -1.0e30)`

See also *lowerObjCutFiniteTrh*

Generic name MSK_DPAR_LOWER_OBJ_CUT

Groups *Termination criteria*

`"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 $-\infty$.

Default -0.5e30

Accepted [-inf; +inf]

Example `M.setSolverParam("lowerObjCutFiniteTrh", -0.5e30)`

Generic name MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH

Groups *Termination criteria*

`"mioMaxTime"`

This parameter limits the maximum time spent by the mixed-integer optimizer. A negative number means infinity.

Default -1.0

Accepted [-inf; +inf]

Example `M.setSolverParam("mioMaxTime", -1.0)`

Generic name MSK_DPAR_MIO_MAX_TIME

Groups *Mixed-integer optimization*, *Termination criteria*

`"mioRelGapConst"`

This value is used to compute the relative gap for the solution to an integer optimization problem.

Default 1.0e-10

Accepted [1.0e-15; +inf]

Example `M.setSolverParam("mioRelGapConst", 1.0e-10)`

Generic name MSK_DPAR_MIO_REL_GAP_CONST

Groups *Mixed-integer optimization*, *Termination criteria*

`"mioTolAbsGap"`

Absolute optimality tolerance employed by the mixed-integer optimizer.

Default 0.0

Accepted [0.0; +inf]

Example `M.setSolverParam("mioTolAbsGap", 0.0)`

Generic name MSK_DPAR_MIO_TOL_ABS_GAP

Groups *Mixed-integer optimization*

`"mioTolAbsRelaxInt"`

Absolute integer feasibility tolerance. If the distance to the nearest integer is less than this tolerance then an integer constraint is assumed to be satisfied.

Default 1.0e-5

Accepted [1e-9; +inf]

Example `M.setSolverParam("mioTolAbsRelaxInt", 1.0e-5)`

Generic name MSK_DPAR_MIO_TOL_ABS_RELAX_INT

Groups *Mixed-integer optimization*

`"mioTolFeas"`

Feasibility tolerance for mixed integer solver.

Default 1.0e-6

Accepted [1e-9; 1e-3]
Example `M.setSolverParam("mioTolFeas", 1.0e-6)`
Generic name MSK_DPAR_MIO_TOL_FEAS
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.

Default 0.0
Accepted [0.0; 1.0]
Example `M.setSolverParam("mioTolRelDualBoundImprovement", 0.0)`
Generic name MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT
Groups *Mixed-integer optimization*

`"mioTolRelGap"`
Relative optimality tolerance employed by the mixed-integer optimizer.

Default 1.0e-4
Accepted [0.0; +inf]
Example `M.setSolverParam("mioTolRelGap", 1.0e-4)`
Generic name MSK_DPAR_MIO_TOL_REL_GAP
Groups *Mixed-integer optimization*, *Termination criteria*

`"optimizerMaxTime"`
Maximum amount of time the optimizer is allowed to spent on the optimization. A negative number means infinity.

Default -1.0
Accepted [-inf; +inf]
Example `M.setSolverParam("optimizerMaxTime", -1.0)`
Generic name MSK_DPAR_OPTIMIZER_MAX_TIME
Groups *Termination criteria*

`"presolveTolAbsLindep"`
Absolute tolerance employed by the linear dependency checker.

Default 1.0e-6
Accepted [0.0; +inf]
Example `M.setSolverParam("presolveTolAbsLindep", 1.0e-6)`
Generic name MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP
Groups *Presolve*

`"presolveTolAij"`
Absolute zero tolerance employed for a_{ij} in the presolve.

Default 1.0e-12
Accepted [1.0e-15; +inf]
Example `M.setSolverParam("presolveTolAij", 1.0e-12)`
Generic name MSK_DPAR_PRESOLVE_TOL_AIJ
Groups *Presolve*

`"presolveTolRelLindep"`
Relative tolerance employed by the linear dependency checker.

Default 1.0e-10
Accepted [0.0; +inf]
Example `M.setSolverParam("presolveTolRelLindep", 1.0e-10)`

Generic name MSK_DPAR_PRESOLVE_TOL_REL_LINDEP
Groups *Presolve*

`"presolveTolS"`
Absolute zero tolerance employed for s_i in the presolve.

Default 1.0e-8
Accepted [0.0; +inf]
Example `M.setSolverParam("presolveTolS", 1.0e-8)`
Generic name MSK_DPAR_PRESOLVE_TOL_S
Groups *Presolve*

`"presolveTolX"`
Absolute zero tolerance employed for x_j in the presolve.

Default 1.0e-8
Accepted [0.0; +inf]
Example `M.setSolverParam("presolveTolX", 1.0e-8)`
Generic name MSK_DPAR_PRESOLVE_TOL_X
Groups *Presolve*

`"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.

Default 0.01
Accepted [1.0e-6; 0.999999]
Example `M.setSolverParam("simLuTolRelPiv", 0.01)`
Generic name MSK_DPAR_SIM_LU_TOL_REL_PIV
Groups *Basis identification*, *Simplex optimizer*

`"simplexAbsTolPiv"`
Absolute pivot tolerance employed by the simplex optimizers.

Default 1.0e-7
Accepted [1.0e-12; +inf]
Example `M.setSolverParam("simplexAbsTolPiv", 1.0e-7)`
Generic name MSK_DPAR_SIMPLEX_ABS_TOL_PIV
Groups *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.

Default 1.0e30
Accepted [-inf; +inf]
Example `M.setSolverParam("upperObjCut", 1.0e30)`
See also *upperObjCutFiniteTrh*
Generic name MSK_DPAR_UPPER_OBJ_CUT
Groups *Termination criteria*

`"upperObjCutFiniteTrh"`
If the upper objective cut is greater than the value of this parameter, then the upper objective cut *upperObjCut* is treated as ∞.

Default 0.5e30
Accepted [-inf; +inf]
Example `M.setSolverParam("upperObjCutFiniteTrh", 0.5e30)`
Generic name MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH
Groups *Termination criteria*

14.4.2 Integer parameters

"autoUpdateSolInfo"

Controls whether the solution information items are automatically updated after an optimization is performed.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("autoUpdateSolInfo", "off")`

Generic name MSK_IPAR_AUTO_UPDATE_SOL_INFO

Groups *Overall system*

"biCleanOptimizer"

Controls which simplex optimizer is used in the clean-up phase. Anything else than *"primalSimplex"* or *"dualSimplex"* is equivalent to *"freeSimplex"*.

Default *"free"*

Accepted *"free"*, *"intpnt"*, *"conic"*, *"primalSimplex"*, *"dualSimplex"*, *"freeSimplex"*, *"mixedInt"*

Example `M.setSolverParam("biCleanOptimizer", "free")`

Generic name MSK_IPAR_BI_CLEAN_OPTIMIZER

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"*.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("biIgnoreMaxIter", "off")`

Generic name MSK_IPAR_BI_IGNORE_MAX_ITER

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"*.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("biIgnoreNumError", "off")`

Generic name MSK_IPAR_BI_IGNORE_NUM_ERROR

Groups *Interior-point method*, *Basis identification*

"biMaxIterations"

Controls the maximum number of simplex iterations allowed to optimize a basis after the basis identification.

Default 1000000

Accepted [0; +inf]

Example `M.setSolverParam("biMaxIterations", 1000000)`

Generic name MSK_IPAR_BI_MAX_ITERATIONS

Groups *Basis identification*, *Termination criteria*

"cacheLicense"

Specifies if the license is kept checked out for the lifetime of the **MOSEK** environment/model/process (*"on"*) or returned to the server immediately after the optimization (*"off"*).

By default the license is checked out for the lifetime of the process by the first call to *Model.solve*.

Check-in and check-out of licenses have an overhead. Frequent communication with the license server should be avoided.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("cacheLicense", "on")`

Generic name MSK_IPAR_CACHE_LICENSE

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.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("infeasPreferPrimal", "on")`

Generic name MSK_IPAR_INFEAS_PREFER_PRIMAL

Groups *Overall solver*

`"intpntBasis"`

Controls whether the interior-point optimizer also computes an optimal basis.

Default *"always"*

Accepted *"never"*, *"always"*, *"noError"*, *"ifFeasible"*, *"reservered"*

Example `M.setSolverParam("intpntBasis", "always")`

See also *biIgnoreMaxIter*, *biIgnoreNumError*, *biMaxIterations*, *biCleanOptimizer*

Generic name MSK_IPAR_INTPNT_BASIS

Groups *Interior-point method*, *Basis identification*

`"intpntDiffStep"`

Controls whether different step sizes are allowed in the primal and dual space.

Default *"on"*

Accepted

- *"on"*: Different step sizes are allowed.
- *"off"*: Different step sizes are not allowed.

Example `M.setSolverParam("intpntDiffStep", "on")`

Generic name MSK_IPAR_INTPNT_DIFF_STEP

Groups *Interior-point method*

`"intpntMaxIterations"`

Controls the maximum number of iterations allowed in the interior-point optimizer.

Default 400

Accepted [0; +inf]

Example `M.setSolverParam("intpntMaxIterations", 400)`

Generic name MSK_IPAR_INTPNT_MAX_ITERATIONS

Groups *Interior-point method*, *Termination criteria*

`"intpntMaxNumCor"`

Controls the maximum number of correctors allowed by the multiple corrector procedure. A negative value means that **MOSEK** is making the choice.

Default -1

Accepted [-1; +inf]

Example `M.setSolverParam("intpntMaxNumCor", -1)`

Generic name MSK_IPAR_INTPNT_MAX_NUM_COR

Groups *Interior-point method*

"intpntMultiThread"

Controls whether the interior-point optimizers are allowed to employ multiple threads if more threads is available.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("intpntMultiThread", "on")`

Generic name MSK_IPAR_INTPNT_MULTI_THREAD

Groups *Overall 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

Default 40

Accepted [0; +inf]

Example `M.setSolverParam("intpntOffColTrh", 40)`

Generic name MSK_IPAR_INTPNT_OFF_COL_TRH

Groups *Interior-point method*

"intpntOrderGpNumSeeds"

The GP ordering is dependent on a random seed. Therefore, trying several random seeds may lead to a better ordering. This parameter controls the number of random seeds tried.

A value of 0 means that MOSEK makes the choice.

Default 0

Accepted [0; +inf]

Example `M.setSolverParam("intpntOrderGpNumSeeds", 0)`

Generic name MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS

Groups *Interior-point method*

"intpntOrderMethod"

Controls the ordering strategy used by the interior-point optimizer when factorizing the Newton equation system.

Default *"free"*

Accepted *"free"*, *"appminloc"*, *"experimental"*, *"tryGraphpar"*, *"forceGraphpar"*, *"none"*

Example `M.setSolverParam("intpntOrderMethod", "free")`

Generic name MSK_IPAR_INTPNT_ORDER_METHOD

Groups *Interior-point method*

"intpntRegularizationUse"

Controls whether regularization is allowed.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("intpntRegularizationUse", "on")`

Generic name MSK_IPAR_INTPNT_REGULARIZATION_USE

Groups *Interior-point method*

"intpntScaling"

Controls how the problem is scaled before the interior-point optimizer is used.

Default *"free"*

Accepted *"free"*, *"none"*, *"moderate"*, *"aggressive"*
Example `M.setSolverParam("intpntScaling", "free")`
Generic name MSK_IPAR_INTPNT_SCALING
Groups *Interior-point method*

`"intpntSolveForm"`
Controls whether the primal or the dual problem is solved.

Default *"free"*
Accepted *"free"*, *"primal"*, *"dual"*
Example `M.setSolverParam("intpntSolveForm", "free")`
Generic name MSK_IPAR_INTPNT_SOLVE_FORM
Groups *Interior-point method*

`"intpntStartingPoint"`
Starting point used by the interior-point optimizer.

Default *"free"*
Accepted *"free"*, *"guess"*, *"constant"*, *"satisfyBounds"*
Example `M.setSolverParam("intpntStartingPoint", "free")`
Generic name MSK_IPAR_INTPNT_STARTING_POINT
Groups *Interior-point method*

`"licenseDebug"`
This option is used to turn on debugging of the license manager.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("licenseDebug", "off")`
Generic name MSK_IPAR_LICENSE_DEBUG
Groups *License manager*

`"licensePauseTime"`
If *licenseWait* is *"on"* and no license is available, then **MOSEK** sleeps a number of milliseconds between each check of whether a license has become free.

Default 100
Accepted [0; 1000000]
Example `M.setSolverParam("licensePauseTime", 100)`
Generic name MSK_IPAR_LICENSE_PAUSE_TIME
Groups *License manager*

`"licenseSuppressExpireWrns"`
Controls whether license features expire warnings are suppressed.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("licenseSuppressExpireWrns", "off")`
Generic name MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS
Groups *License manager*, *Output information*

`"licenseTrhExpiryWrn"`
If a license feature expires in a numbers of days less than the value of this parameter then a warning will be issued.

Default 7
Accepted [0; +inf]
Example `M.setSolverParam("licenseTrhExpiryWrn", 7)`

Generic name MSK_IPAR_LICENSE_TRH_EXPIRY_WRN
Groups *License manager*, *Output information*

`"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.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("licenseWait", "off")`
Generic name MSK_IPAR_LICENSE_WAIT
Groups *Overall solver*, *Overall 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.

Default 10
Accepted [0; +inf]
Example `M.setSolverParam("log", 10)`
See also *logCutSecondOpt*
Generic name MSK_IPAR_LOG
Groups *Output information*, *Logging*

`"logBi"`
Controls the amount of output printed by the basis identification procedure. A higher level implies that more information is logged.

Default 1
Accepted [0; +inf]
Example `M.setSolverParam("logBi", 1)`
Generic name MSK_IPAR_LOG_BI
Groups *Basis identification*, *Output information*, *Logging*

`"logBiFreq"`
Controls how frequently the optimizer outputs information about the basis identification and how frequent the user-defined callback function is called.

Default 2500
Accepted [0; +inf]
Example `M.setSolverParam("logBiFreq", 2500)`
Generic name MSK_IPAR_LOG_BI_FREQ
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.

Default 1
Accepted [0; +inf]
Example `M.setSolverParam("logCutSecondOpt", 1)`
See also *log*, *logIntpnt*, *logMio*, *logSim*
Generic name MSK_IPAR_LOG_CUT_SECOND_OPT
Groups *Output information*, *Logging*

"logExpand"

Controls the amount of logging when a data item such as the maximum number constrains is expanded.

Default 0

Accepted [0; +inf]

Example `M.setSolverParam("logExpand", 0)`

Generic name MSK_IPAR_LOG_EXPAND

Groups *Output information*, *Logging*

"logFile"

If turned on, then some log info is printed when a file is written or read.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logFile", 1)`

Generic name MSK_IPAR_LOG_FILE

Groups *Data input/output*, *Output information*, *Logging*

"logInfeasAna"

Controls amount of output printed by the infeasibility analyzer procedures. A higher level implies that more information is logged.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logInfeasAna", 1)`

Generic name MSK_IPAR_LOG_INFEAS_ANA

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.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logIntpnt", 1)`

Generic name MSK_IPAR_LOG_INTPNT

Groups *Interior-point method*, *Output information*, *Logging*

"logLocalInfo"

Controls whether local identifying information like environment variables, filenames, IP addresses etc. are printed to the log.

Note that this will only affect some functions. Some functions that specifically emit system information will not be affected.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("logLocalInfo", "on")`

Generic name MSK_IPAR_LOG_LOCAL_INFO

Groups *Output information*, *Logging*

"logMio"

Controls the log level for the mixed-integer optimizer. A higher level implies that more information is logged.

Default 4

Accepted [0; +inf]

Example `M.setSolverParam("logMio", 4)`

Generic name MSK_IPAR_LOG_MIO

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.

Default 10

Accepted [-inf; +inf]

Example `M.setSolverParam("logMioFreq", 10)`

Generic name MSK_IPAR_LOG_MIO_FREQ

Groups *Mixed-integer optimization*, *Output information*, *Logging*

"logOrder"

If turned on, then factor lines are added to the log.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logOrder", 1)`

Generic name MSK_IPAR_LOG_ORDER

Groups *Output information*, *Logging*

"logPresolve"

Controls amount of output printed by the presolve procedure. A higher level implies that more information is logged.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logPresolve", 1)`

Generic name MSK_IPAR_LOG_PRESOLVE

Groups *Logging*

"logResponse"

Controls amount of output printed when response codes are reported. A higher level implies that more information is logged.

Default 0

Accepted [0; +inf]

Example `M.setSolverParam("logResponse", 0)`

Generic name MSK_IPAR_LOG_RESPONSE

Groups *Output information*, *Logging*

"logSim"

Controls amount of output printed by the simplex optimizer. A higher level implies that more information is logged.

Default 4

Accepted [0; +inf]

Example `M.setSolverParam("logSim", 4)`

Generic name MSK_IPAR_LOG_SIM

Groups *Simplex optimizer*, *Output information*, *Logging*

"logSimFreq"

Controls how frequent the simplex optimizer outputs information about the optimization and how frequent the user-defined callback function is called.

Default 1000

Accepted [0; +inf]

Example `M.setSolverParam("logSimFreq", 1000)`

Generic name MSK_IPAR_LOG_SIM_FREQ

Groups *Simplex optimizer*, *Output information*, *Logging*

`"logSimMinor"`
Currently not in use.

Default 1

Accepted [0; +inf]

Example `M.setSolverParam("logSimMinor", 1)`

Generic name MSK_IPAR_LOG_SIM_MINOR

Groups *Simplex optimizer*, *Output information*

`"mioBranchDir"`
Controls whether the mixed-integer optimizer is branching up or down by default.

Default *"free"*

Accepted *"free"*, *"up"*, *"down"*, *"near"*, *"far"*, *"rootLp"*, *"guided"*, *"pseudocost"*

Example `M.setSolverParam("mioBranchDir", "free")`

Generic name MSK_IPAR_MIO_BRANCH_DIR

Groups *Mixed-integer optimization*

`"mioConicOuterApproximation"`
If this option is turned on outer approximation is used when solving relaxations of conic problems; otherwise interior point is used.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioConicOuterApproximation", "off")`

Generic name MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION

Groups *Mixed-integer optimization*

`"mioCutClique"`
Controls whether clique cuts should be generated.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioCutClique", "on")`

Generic name MSK_IPAR_MIO_CUT_CLIQUE

Groups *Mixed-integer optimization*

`"mioCutCmir"`
Controls whether mixed integer rounding cuts should be generated.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioCutCmir", "on")`

Generic name MSK_IPAR_MIO_CUT_CMIR

Groups *Mixed-integer optimization*

`"mioCutGmi"`
Controls whether GMI cuts should be generated.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioCutGmi", "on")`

Generic name MSK_IPAR_MIO_CUT_GMI

Groups *Mixed-integer optimization*

"mioCutImpliedBound"
Controls whether implied bound cuts should be generated.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("mioCutImpliedBound", "off")`
Generic name MSK_IPAR_MIO_CUT_IMPLIED_BOUND
Groups *Mixed-integer optimization*

"mioCutKnapsackCover"
Controls whether knapsack cover cuts should be generated.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("mioCutKnapsackCover", "off")`
Generic name MSK_IPAR_MIO_CUT_KNAPSACK_COVER
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

Default -1
Accepted [-1; +1]
Example `M.setSolverParam("mioCutSelectionLevel", -1)`
Generic name MSK_IPAR_MIO_CUT_SELECTION_LEVEL
Groups *Mixed-integer optimization*

"mioFeaspumpLevel"
Controls the way the Feasibility Pump heuristic is employed by the mixed-integer optimizer.

- -1. The optimizer chooses how the Feasibility Pump is used
- 0. The Feasibility Pump is disabled
- 1. The Feasibility Pump is enabled with an effort to improve solution quality
- 2. The Feasibility Pump is enabled with an effort to reach feasibility early

Default -1
Accepted [-1; 2]
Example `M.setSolverParam("mioFeaspumpLevel", -1)`
Generic name MSK_IPAR_MIO_FEASPUMP_LEVEL
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.

Default -1
Accepted [-inf; +inf]
Example `M.setSolverParam("mioHeuristicLevel", -1)`
Generic name MSK_IPAR_MIO_HEURISTIC_LEVEL
Groups *Mixed-integer optimization*

"mioMaxNumBranches"

Maximum number of branches allowed during the branch and bound search. A negative value means infinite.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("mioMaxNumBranches", -1)`

Generic name MSK_IPAR_MIO_MAX_NUM_BRANCHES

Groups *Mixed-integer optimization*, *Termination criteria*

"mioMaxNumRelaxs"

Maximum number of relaxations allowed during the branch and bound search. A negative value means infinite.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("mioMaxNumRelaxs", -1)`

Generic name MSK_IPAR_MIO_MAX_NUM_RELAXS

Groups *Mixed-integer optimization*

"mioMaxNumRootCutRounds"

Maximum number of cut separation rounds at the root node.

Default 100

Accepted [0; +inf]

Example `M.setSolverParam("mioMaxNumRootCutRounds", 100)`

Generic name MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS

Groups *Mixed-integer optimization*, *Termination criteria*

"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.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("mioMaxNumSolutions", -1)`

Generic name MSK_IPAR_MIO_MAX_NUM_SOLUTIONS

Groups *Mixed-integer optimization*, *Termination criteria*

"mioMode"

Controls whether the optimizer includes the integer restrictions when solving a (mixed) integer optimization problem.

Default *"satisfied"*

Accepted *"ignored"*, *"satisfied"*

Example `M.setSolverParam("mioMode", "satisfied")`

Generic name MSK_IPAR_MIO_MODE

Groups *Overall solver*

"mioNodeOptimizer"

Controls which optimizer is employed at the non-root nodes in the mixed-integer optimizer.

Default *"free"*

Accepted *"free"*, *"intpnt"*, *"conic"*, *"primalSimplex"*, *"dualSimplex"*, *"freeSimplex"*, *"mixedInt"*

Example `M.setSolverParam("mioNodeOptimizer", "free")`

Generic name MSK_IPAR_MIO_NODE_OPTIMIZER

Groups *Mixed-integer optimization*

"mioNodeSelection"

Controls the node selection strategy employed by the mixed-integer optimizer.

Default *"free"*

Accepted *"free"*, *"first"*, *"best"*, *"pseudo"*

Example `M.setSolverParam("mioNodeSelection", "free")`

Generic name MSK_IPAR_MIO_NODE_SELECTION

Groups *Mixed-integer optimization*

"mioPerspectiveReformulate"

Enables or disables perspective reformulation in presolve.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioPerspectiveReformulate", "on")`

Generic name MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE

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

Default -1

Accepted [-1; 3]

Example `M.setSolverParam("mioProbingLevel", -1)`

Generic name MSK_IPAR_MIO_PROBING_LEVEL

Groups *Mixed-integer optimization*

"mioPropagateObjectiveConstraint"

Use objective domain propagation.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("mioPropagateObjectiveConstraint", "off")`

Generic name MSK_IPAR_MIO_PROPAGATE_OBJECTIVE_CONSTRAINT

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.

Default -1

Accepted [-1; +inf]

Example `M.setSolverParam("mioRinsMaxNodes", -1)`

Generic name MSK_IPAR_MIO_RINS_MAX_NODES

Groups *Mixed-integer optimization*

"mioRootOptimizer"

Controls which optimizer is employed at the root node in the mixed-integer optimizer.

Default *"free"*

Accepted *"free"*, *"intpnt"*, *"conic"*, *"primalSimplex"*, *"dualSimplex"*, *"freeSimplex"*, *"mixedInt"*

Example `M.setSolverParam("mioRootOptimizer", "free")`

Generic name MSK_IPAR_MIO_ROOT_OPTIMIZER

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

Default -1

Accepted [-1; 1]

Example `M.setSolverParam("mioRootRepeatPresolveLevel", -1)`

Generic name MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL

Groups *Mixed-integer optimization*

`"mioSeed"`

Sets the random seed used for randomization in the mixed integer optimizer. Selecting a different seed can change the path the optimizer takes to the optimal solution.

Default 42

Accepted [0; +inf]

Example `M.setSolverParam("mioSeed", 42)`

Generic name MSK_IPAR_MIO_SEED

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

Default -1

Accepted [-1; +2]

Example `M.setSolverParam("mioVbDetectionLevel", -1)`

Generic name MSK_IPAR_MIO_VB_DETECTION_LEVEL

Groups *Mixed-integer optimization*

`"mtSpincount"`

Set the number of iterations to spin before sleeping.

Default 0

Accepted [0; 1000000000]

Example `M.setSolverParam("mtSpincount", 0)`

Generic name MSK_IPAR_MT_SPINCOUNT

Groups *Overall 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.

If using the conic optimizer, the value of this parameter set at first optimization remains constant through the lifetime of the process. **MOSEK** will allocate a thread pool of given size, and changing the parameter value later will have no effect. It will, however, remain possible to demand single-threaded execution by setting *intpntMultiThread*.

For the mixed-integer optimizer and interior-point linear optimizer there is no such restriction.

Default 0

Accepted [0; +inf]

Example `M.setSolverParam("numThreads", 0)`

Generic name MSK_IPAR_NUM_THREADS

Groups *Overall system*

`"optimizer"`

The parameter controls which optimizer is used to optimize the task.

Default *"free"*

Accepted *"free"*, *"intpnt"*, *"conic"*, *"primalSimplex"*, *"dualSimplex"*, *"freeSimplex"*, *"mixedInt"*

Example `M.setSolverParam("optimizer", "free")`

Generic name MSK_IPAR_OPTIMIZER

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 means the parameter value is selected automatically.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("presolveEliminatorMaxFill", -1)`

Generic name MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL

Groups *Presolve*

`"presolveEliminatorMaxNumTries"`

Control the maximum number of times the eliminator is tried. A negative value implies **MOSEK** decides.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("presolveEliminatorMaxNumTries", -1)`

Generic name MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES

Groups *Presolve*

`"presolveLevel"`

Currently not used.

Default -1

Accepted [-inf; +inf]

Example `M.setSolverParam("presolveLevel", -1)`

Generic name MSK_IPAR_PRESOLVE_LEVEL

Groups *Overall solver*, *Presolve*

`"presolveLindepAbsWorkTrh"`

Controls linear dependency check in presolve. The linear dependency check is potentially computationally expensive.

Default 100

Accepted [-inf; +inf]

Example `M.setSolverParam("presolveLindepAbsWorkTrh", 100)`

Generic name MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH

Groups *Presolve*

`"presolveLindepRelWorkTrh"`

Controls linear dependency check in presolve. The linear dependency check is potentially computationally expensive.

Default 100
Accepted [-inf; +inf]
Example `M.setSolverParam("presolveLindepRelWorkTrh", 100)`
Generic name MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH
Groups *Presolve*

`"presolveLindepUse"`
Controls whether the linear constraints are checked for linear dependencies.

Default *"on"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("presolveLindepUse", "on")`
Generic name MSK_IPAR_PRESOLVE_LINDEP_USE
Groups *Presolve*

`"presolveMaxNumPass"`
Control the maximum number of times presolve passes over the problem. A negative value implies **MOSEK** decides.

Default -1
Accepted [-inf; +inf]
Example `M.setSolverParam("presolveMaxNumPass", -1)`
Generic name MSK_IPAR_PRESOLVE_MAX_NUM_PASS
Groups *Presolve*

`"presolveUse"`
Controls whether the presolve is applied to a problem before it is optimized.

Default *"free"*
Accepted *"off"*, *"on"*, *"free"*
Example `M.setSolverParam("presolveUse", "free")`
Generic name MSK_IPAR_PRESOLVE_USE
Groups *Overall solver*, *Presolve*

`"removeUnusedSolutions"`
Removes unused solutions before the optimization is performed.

Default *"off"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("removeUnusedSolutions", "off")`
Generic name MSK_IPAR_REMOVE_UNUSED_SOLUTIONS
Groups *Overall system*

`"simBasisFactorUse"`
Controls whether an 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.

Default *"on"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("simBasisFactorUse", "on")`
Generic name MSK_IPAR_SIM_BASIS_FACTOR_USE
Groups *Simplex optimizer*

`"simDegen"`
Controls how aggressively degeneration is handled.

Default *"free"*
Accepted *"none"*, *"free"*, *"aggressive"*, *"moderate"*, *"minimum"*

Example `M.setSolverParam("simDegen", "free")`

Generic name MSK_IPAR_SIM_DEGEN

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.

Default 90

Accepted [0; +inf]

Example `M.setSolverParam("simDualCrash", 90)`

Generic name MSK_IPAR_SIM_DUAL_CRASH

Groups *Dual simplex*

`"simDualPhaseoneMethod"`

An experimental feature.

Default 0

Accepted [0; 10]

Example `M.setSolverParam("simDualPhaseoneMethod", 0)`

Generic name MSK_IPAR_SIM_DUAL_PHASEONE_METHOD

Groups *Simplex optimizer*

`"simDualRestrictSelection"`

The dual simplex optimizer can use a so-called restricted selection/pricing strategy to choose 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.

Default 50

Accepted [0; 100]

Example `M.setSolverParam("simDualRestrictSelection", 50)`

Generic name MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION

Groups *Dual simplex*

`"simDualSelection"`

Controls the choice of the incoming variable, known as the selection strategy, in the dual simplex optimizer.

Default *"free"*

Accepted *"free"*, *"full"*, *"ase"*, *"devex"*, *"se"*, *"partial"*

Example `M.setSolverParam("simDualSelection", "free")`

Generic name MSK_IPAR_SIM_DUAL_SELECTION

Groups *Dual simplex*

`"simExploitDupvec"`

Controls if the simplex optimizers are allowed to exploit duplicated columns.

Default *"off"*

Accepted *"on"*, *"off"*, *"free"*

Example `M.setSolverParam("simExploitDupvec", "off")`

Generic name MSK_IPAR_SIM_EXPLOIT_DUPVEC

Groups *Simplex optimizer*

`"simHotstart"`

Controls the type of hot-start that the simplex optimizer perform.

Default *"free"*
Accepted *"none"*, *"free"*, *"statusKeys"*
Example `M.setSolverParam("simHotstart", "free")`
Generic name MSK_IPAR_SIM_HOTSTART
Groups *Simplex optimizer*

`"simHotstartLu"`
Determines if the simplex optimizer should exploit the initial factorization.

Default *"on"*
Accepted

- *"on"*: Factorization is reused if possible.
- *"off"*: Factorization is recomputed.

Example `M.setSolverParam("simHotstartLu", "on")`
Generic name MSK_IPAR_SIM_HOTSTART_LU
Groups *Simplex optimizer*

`"simMaxIterations"`
Maximum number of iterations that can be used by a simplex optimizer.

Default 10000000
Accepted [0; +inf]
Example `M.setSolverParam("simMaxIterations", 10000000)`
Generic name MSK_IPAR_SIM_MAX_ITERATIONS
Groups *Simplex optimizer*, *Termination criteria*

`"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.

Default 250
Accepted [0; +inf]
Example `M.setSolverParam("simMaxNumSetbacks", 250)`
Generic name MSK_IPAR_SIM_MAX_NUM_SETBACKS
Groups *Simplex optimizer*

`"simNonSingular"`
Controls if the simplex optimizer ensures a non-singular basis, if possible.

Default *"on"*
Accepted *"on"*, *"off"*
Example `M.setSolverParam("simNonSingular", "on")`
Generic name MSK_IPAR_SIM_NON_SINGULAR
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.

Default 90
Accepted [0; +inf]
Example `M.setSolverParam("simPrimalCrash", 90)`
Generic name MSK_IPAR_SIM_PRIMAL_CRASH
Groups *Primal simplex*

`"simPrimalPhaseoneMethod"`
An experimental feature.

Default 0

Accepted [0; 10]

Example `M.setSolverParam("simPrimalPhaseoneMethod", 0)`

Generic name MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD

Groups *Simplex optimizer*

`"simPrimalRestrictSelection"`

The primal simplex optimizer can use a so-called restricted selection/pricing strategy to choose 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.

Default 50

Accepted [0; 100]

Example `M.setSolverParam("simPrimalRestrictSelection", 50)`

Generic name MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION

Groups *Primal simplex*

`"simPrimalSelection"`

Controls the choice of the incoming variable, known as the selection strategy, in the primal simplex optimizer.

Default *"free"*

Accepted *"free"*, *"full"*, *"ase"*, *"devex"*, *"se"*, *"partial"*

Example `M.setSolverParam("simPrimalSelection", "free")`

Generic name MSK_IPAR_SIM_PRIMAL_SELECTION

Groups *Primal simplex*

`"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.

Default 0

Accepted [0; +inf]

Example `M.setSolverParam("simRefactorFreq", 0)`

Generic name MSK_IPAR_SIM_REFACTOR_FREQ

Groups *Simplex optimizer*

`"simReformulation"`

Controls if the simplex optimizers are allowed to reformulate the problem.

Default *"off"*

Accepted *"on"*, *"off"*, *"free"*, *"aggressive"*

Example `M.setSolverParam("simReformulation", "off")`

Generic name MSK_IPAR_SIM_REFORMULATION

Groups *Simplex optimizer*

`"simSaveLu"`

Controls if the LU factorization stored should be replaced with the LU factorization corresponding to the initial basis.

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("simSaveLu", "off")`

Generic name MSK_IPAR_SIM_SAVE_LU

Groups *Simplex optimizer*

"simScaling"

Controls how much effort is used in scaling the problem before a simplex optimizer is used.

Default *"free"*

Accepted *"free"*, *"none"*, *"moderate"*, *"aggressive"*

Example `M.setSolverParam("simScaling", "free")`

Generic name MSK_IPAR_SIM_SCALING

Groups *Simplex optimizer*

"simScalingMethod"

Controls how the problem is scaled before a simplex optimizer is used.

Default *"pow2"*

Accepted *"pow2"*, *"free"*

Example `M.setSolverParam("simScalingMethod", "pow2")`

Generic name MSK_IPAR_SIM_SCALING_METHOD

Groups *Simplex optimizer*

"simSeed"

Sets the random seed used for randomization in the simplex optimizers.

Default 23456

Accepted [0; 32749]

Example `M.setSolverParam("simSeed", 23456)`

Generic name MSK_IPAR_SIM_SEED

Groups *Simplex optimizer*

"simSolveForm"

Controls whether the primal or the dual problem is solved by the primal-/dual-simplex optimizer.

Default *"free"*

Accepted *"free"*, *"primal"*, *"dual"*

Example `M.setSolverParam("simSolveForm", "free")`

Generic name MSK_IPAR_SIM_SOLVE_FORM

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).

Default *"off"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("simSwitchOptimizer", "off")`

Generic name MSK_IPAR_SIM_SWITCH_OPTIMIZER

Groups *Simplex optimizer*

"writeLpFullObj"

Write all variables, including the ones with 0-coefficients, in the objective.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("writeLpFullObj", "on")`

Generic name MSK_IPAR_WRITE_LP_FULL_OBJ

Groups *Data input/output*

"writeLpLineWidth"

Maximum width of line in an LP file written by **MOSEK**.

Default 80

Accepted [40; +inf]

Example `M.setSolverParam("writeLpLineWidth", 80)`

Generic name MSK_IPAR_WRITE_LP_LINE_WIDTH

Groups *Data input/output*

"writeLpQuotedNames"

If this option is turned on, then **MOSEK** will quote invalid LP names when writing an LP file.

Default *"on"*

Accepted *"on"*, *"off"*

Example `M.setSolverParam("writeLpQuotedNames", "on")`

Generic name MSK_IPAR_WRITE_LP_QUOTED_NAMES

Groups *Data input/output*

"writeLpTermsPerLine"

Maximum number of terms on a single line in an LP file written by **MOSEK**. 0 means unlimited.

Default 10

Accepted [0; +inf]

Example `M.setSolverParam("writeLpTermsPerLine", 10)`

Generic name MSK_IPAR_WRITE_LP_TERMS_PER_LINE

Groups *Data input/output*

14.4.3 String parameters

"basSolFileName"

Name of the `bas` solution file.

Accepted Any valid file name.

Example `M.setSolverParam("basSolFileName", "somevalue")`

Generic name MSK_SPAR_BAS_SOL_FILE_NAME

Groups *Data input/output*, *Solution input/output*

"dataFileName"

Data are read and written to this file.

Accepted Any valid file name.

Example `M.setSolverParam("dataFileName", "somevalue")`

Generic name MSK_SPAR_DATA_FILE_NAME

Groups *Data input/output*

"intSolFileName"

Name of the `int` solution file.

Accepted Any valid file name.

Example `M.setSolverParam("intSolFileName", "somevalue")`

Generic name MSK_SPAR_INT_SOL_FILE_NAME

Groups *Data input/output*, *Solution input/output*

"itrSolFileName"

Name of the `itr` solution file.

Accepted Any valid file name.

Example `M.setSolverParam("itrSolFileName", "somevalue")`

Generic name MSK_SPAR_ITR_SOL_FILE_NAME

Groups *Data input/output*, *Solution 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 Any valid string.

Example `M.setSolverParam("remoteAccessToken", "somevalue")`

Generic name MSK_SPAR_REMOTE_ACCESS_TOKEN

Groups *Overall system*

`"writeLpGenVarName"`

Sometimes when an LP file is written additional variables must be inserted. They will have the prefix denoted by this parameter.

Default xmskgen

Accepted Any valid string.

Example `M.setSolverParam("writeLpGenVarName", "xmskgen")`

Generic name MSK_SPAR_WRITE_LP_GEN_VAR_NAME

Groups *Data input/output*

14.5 Enumerations

`AccSolutionStatus`

Constants used for defining which solutions statuses are acceptable.

`Anything`

Accept all solution status except *`SolutionStatus.Undefined`*.

`Optimal`

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.

`ProblemStatus`

Constants defining the problem status.

`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.

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 or the solver stalled.

Optimal

The solution values are feasible and optimal.

Feasible

The solution is feasible.

Certificate

The solution is a certificate of infeasibility (primal or dual, depending on which solution it belongs to).

IllposedCert

The solution is a certificate of illposedness.

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.

14.6 Constants

14.6.1 Basis identification

"never"

Never do basis identification.

"always"

Basis identification is always performed even if the interior-point optimizer terminates abnormally.

"noError"

Basis identification is performed if the interior-point optimizer terminates without an error.

"ifFeasible"

Basis identification is not performed if the interior-point optimizer terminates with a problem status saying that the problem is primal or dual infeasible.

"reservered"

Not currently in use.

14.6.2 Bound keys

`"lo"`
: The constraint or variable has a finite lower bound and an infinite upper bound.

`"up"`
: The constraint or variable has an infinite lower bound and an finite upper bound.

`"fx"`
: The constraint or variable is fixed.

`"fr"`
: The constraint or variable is free.

`"ra"`
: The constraint or variable is ranged.

14.6.3 Mark

`"lo"`
: The lower bound is selected for sensitivity analysis.

`"up"`
: The upper bound is selected for sensitivity analysis.

14.6.4 Degeneracy strategies

`"none"`
: The simplex optimizer should use no degeneration strategy.

`"free"`
: The simplex optimizer chooses the degeneration strategy.

`"aggressive"`
: The simplex optimizer should use an aggressive degeneration strategy.

`"moderate"`
: The simplex optimizer should use a moderate degeneration strategy.

`"minimum"`
: The simplex optimizer should use a minimum degeneration strategy.

14.6.5 Transposed matrix.

`"no"`
: No transpose is applied.

`"yes"`
: A transpose is applied.

14.6.6 Triangular part of a symmetric matrix.

`"lo"`
: Lower part.

`"up"`
: Upper part.

14.6.7 Problem reformulation.

`"on"`
: Allow the simplex optimizer to reformulate the problem.

`"off"`
: Disallow the simplex optimizer to reformulate the problem.

`"free"`
: The simplex optimizer can choose freely.

`"aggressive"`
: The simplex optimizer should use an aggressive reformulation strategy.

14.6.8 Exploit duplicate columns.

"on"
: Allow the simplex optimizer to exploit duplicated columns.

"off"
: Disallow the simplex optimizer to exploit duplicated columns.

"free"
: The simplex optimizer can choose freely.

14.6.9 Hot-start type employed by the simplex optimizer

"none"
: The simplex optimizer performs a coldstart.

"free"
: The simplex optimize chooses the hot-start type.

"statusKeys"
: Only the status keys of the constraints and variables are used to choose the type of hot-start.

14.6.10 Hot-start type employed by the interior-point optimizers.

"none"
: The interior-point optimizer performs a coldstart.

"primal"
: The interior-point optimizer exploits the primal solution only.

"dual"
: The interior-point optimizer exploits the dual solution only.

"primalDual"
: The interior-point optimizer exploits both the primal and dual solution.

14.6.11 Solution purification employed optimizer.

"none"
: The optimizer performs no solution purification.

"primal"
: The optimizer purifies the primal solution.

"dual"
: The optimizer purifies the dual solution.

"primalDual"
: The optimizer purifies both the primal and dual solution.

"auto"
: TBD

14.6.12 Progress callback codes

"beginBi"
: The basis identification procedure has been started.

"beginConic"
: The callback function is called when the conic optimizer is started.

"beginDualBi"
: The callback function is called from within the basis identification procedure when the dual phase is started.

"beginDualSensitivity"
: Dual sensitivity analysis is started.

"beginDualSetupBi"
: The callback function is called when the dual BI phase is started.

"beginDualSimplex"
: The callback function is called when the dual simplex optimizer started.

"beginDualSimplexBi"
: The callback function is called from within the basis identification procedure when the dual simplex clean-up phase is started.

"beginFullConvexityCheck"
: Begin full convexity check.

"beginInfeasAna"
: The callback function is called when the infeasibility analyzer is started.

"beginIntpnt"
: The callback function is called when the interior-point optimizer is started.

"beginLicenseWait"
: Begin waiting for license.

"beginMio"
: The callback function is called when the mixed-integer optimizer is started.

"beginOptimizer"
: The callback function is called when the optimizer is started.

"beginPresolve"
: The callback function is called when the presolve is started.

"beginPrimalBi"
: The callback function is called from within the basis identification procedure when the primal phase is started.

"beginPrimalRepair"
: Begin primal feasibility repair.

"beginPrimalSensitivity"
: Primal sensitivity analysis is started.

"beginPrimalSetupBi"
: The callback function is called when the primal BI setup is started.

"beginPrimalSimplex"
: The callback function is called when the primal simplex optimizer is started.

"beginPrimalSimplexBi"
: The callback function is called from within the basis identification procedure when the primal simplex clean-up phase is started.

"beginQcqoReformulate"
: Begin QCQO reformulation.

"beginRead"
: **MOSEK** has started reading a problem file.

"beginRootCutgen"
: The callback function is called when root cut generation is started.

"beginSimplex"
: The callback function is called when the simplex optimizer is started.

"beginSimplexBi"
: The callback function is called from within the basis identification procedure when the simplex clean-up phase is started.

"beginToConic"
: Begin conic reformulation.

"beginWrite"
: **MOSEK** has started writing a problem file.

"conic"
: The callback function is called from within the conic optimizer after the information database has been updated.

"dualSimplex"
: The callback function is called from within the dual simplex optimizer.

"endBi"
: The callback function is called when the basis identification procedure is terminated.

"endConic"
: The callback function is called when the conic optimizer is terminated.

"endDualBi"
: The callback function is called from within the basis identification procedure when the dual phase is terminated.

"endDualSensitivity"
: Dual sensitivity analysis is terminated.

"endDualSetupBi"
: The callback function is called when the dual BI phase is terminated.

"endDualSimplex"
: The callback function is called when the dual simplex optimizer is terminated.

"endDualSimplexBi"
: The callback function is called from within the basis identification procedure when the dual clean-up phase is terminated.

"endFullConvexityCheck"
: End full convexity check.

"endInfeasAna"
: The callback function is called when the infeasibility analyzer is terminated.

"endIntpnt"
: The callback function is called when the interior-point optimizer is terminated.

"endLicenseWait"
: End waiting for license.

"endMio"
: The callback function is called when the mixed-integer optimizer is terminated.

"endOptimizer"
: The callback function is called when the optimizer is terminated.

"endPresolve"
: The callback function is called when the presolve is completed.

"endPrimalBi"
: The callback function is called from within the basis identification procedure when the primal phase is terminated.

"endPrimalRepair"
: End primal feasibility repair.

"endPrimalSensitivity"
: Primal sensitivity analysis is terminated.

"endPrimalSetupBi"
: The callback function is called when the primal BI setup is terminated.

"endPrimalSimplex"
: The callback function is called when the primal simplex optimizer is terminated.

"endPrimalSimplexBi"
: The callback function is called from within the basis identification procedure when the primal clean-up phase is terminated.

"endQcqoReformulate"
: End QCQO reformulation.

"endRead"
: **MOSEK** has finished reading a problem file.

"endRootCutgen"
: The callback function is called when root cut generation is terminated.

"endSimplex"
: The callback function is called when the simplex optimizer is terminated.

"endSimplexBi"
: The callback function is called from within the basis identification procedure when the simplex clean-up phase is terminated.

"endToConic"
: End conic reformulation.

"endWrite"
: **MOSEK** has finished writing a problem file.

"imBi"
: The callback function is called from within the basis identification procedure at an intermediate point.

"imConic"
: The callback function is called at an intermediate stage within the conic optimizer where the information database has not been updated.

"imDualBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

"imDualSensivity"
: The callback function is called at an intermediate stage of the dual sensitivity analysis.

"imDualSimplex"
: The callback function is called at an intermediate point in the dual simplex optimizer.

"imFullConvexityCheck"
: The callback function is called at an intermediate stage of the full convexity check.

"imIntpnt"
: The callback function is called at an intermediate stage within the interior-point optimizer where the information database has not been updated.

"imLicenseWait"
: **MOSEK** is waiting for a license.

"imLu"
: The callback function is called from within the LU factorization procedure at an intermediate point.

"imMio"
: The callback function is called at an intermediate point in the mixed-integer optimizer.

"imMioDualSimplex"
: The callback function is called at an intermediate point in the mixed-integer optimizer while running the dual simplex optimizer.

"imMioIntpnt"
: The callback function is called at an intermediate point in the mixed-integer optimizer while running the interior-point optimizer.

"imMioPrimalSimplex"
: The callback function is called at an intermediate point in the mixed-integer optimizer while running the primal simplex optimizer.

"imOrder"
: The callback function is called from within the matrix ordering procedure at an intermediate point.

"imPresolve"
: The callback function is called from within the presolve procedure at an intermediate stage.

"imPrimalBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

"imPrimalSensivity"
: The callback function is called at an intermediate stage of the primal sensitivity analysis.

"imPrimalSimplex"
: The callback function is called at an intermediate point in the primal simplex optimizer.

"imQoReformulate"
: The callback function is called at an intermediate stage of the conic quadratic reformulation.

"imRead"
: Intermediate stage in reading.

"imRootCutgen"
: The callback is called from within root cut generation at an intermediate stage.

"imSimplex"
: The callback function is called from within the simplex optimizer at an intermediate point.

"imSimplexBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the simplex clean-up phase. The frequency of the callbacks is controlled by the *logSimFreq* parameter.

"intpnt"
: The callback function is called from within the interior-point optimizer after the information database has been updated.

"newIntMio"
: The callback function is called after a new integer solution has been located by the mixed-integer optimizer.

"primalSimplex"
: The callback function is called from within the primal simplex optimizer.

"readOpf"
: The callback function is called from the OPF reader.

"readOpfSection"
: A chunk of Q non-zeros has been read from a problem file.

"solvingRemote"
: The callback function is called while the task is being solved on a remote server.

"updateDualBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

"updateDualSimplex"
: The callback function is called in the dual simplex optimizer.

"updateDualSimplexBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the dual simplex clean-up phase. The frequency of the callbacks is controlled by the *logSimFreq* parameter.

"updatePresolve"
: The callback function is called from within the presolve procedure.

"updatePrimalBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

"updatePrimalSimplex"
: The callback function is called in the primal simplex optimizer.

"updatePrimalSimplexBi"
: The callback function is called from within the basis identification procedure at an intermediate point in the primal simplex clean-up phase. The frequency of the callbacks is controlled by the *logSimFreq* parameter.

"writeOpf"
: The callback function is called from the OPF writer.

14.6.13 Types of convexity checks.

"none"
: No convexity check.

"simple"
: Perform simple and fast convexity check.

"full"
: Perform a full convexity check.

14.6.14 Compression types

"none"
: No compression is used.

"free"
: The type of compression used is chosen automatically.

"gzip"
: The type of compression used is gzip compatible.

"zstd"
: The type of compression used is zstd compatible.

14.6.15 Cone types

"quad"
: The cone is a quadratic cone.

"rquad"
: The cone is a rotated quadratic cone.

"pexp"
: A primal exponential cone.

"dexp"
: A dual exponential cone.

"ppow"
: A primal power cone.

"dpow"
: A dual power cone.

"zero"
: The zero cone.

14.6.16 Name types

"gen"
: General names. However, no duplicate and blank names are allowed.

"mps"
: MPS type names.

"lp"
: LP type names.

14.6.17 SCopt operator types

"ent"
: Entropy

"exp"
: Exponential

"log"
: Logarithm

"pow"
: Power

"sqrt"
: Square root

14.6.18 Cone types

"sparse"
: Sparse symmetric matrix.

14.6.19 Data format types

"extension"
: The file extension is used to determine the data file format.

"mps"
: The data file is MPS formatted.

"lp"
: The data file is LP formatted.

"op"
: The data file is an optimization problem formatted file.

"freeMps"
: The data a free MPS formatted file.

"task"
: Generic task dump file.

"ptf"
: (P)retty (T)ext (F)format.

"cb"
: Conic benchmark format,

"jsonTask"
: JSON based task format.

14.6.20 Double information items

"biCleanDualTime"
: Time spent within the dual clean-up optimizer of the basis identification procedure since its invocation.

"biCleanPrimalTime"
: Time spent within the primal clean-up optimizer of the basis identification procedure since its invocation.

"biCleanTime"
: Time spent within the clean-up phase of the basis identification procedure since its invocation.

"biDualTime"
: Time spent within the dual phase basis identification procedure since its invocation.

"biPrimalTime"
: Time spent within the primal phase of the basis identification procedure since its invocation.

"biTime"
: Time spent within the basis identification procedure since its invocation.

"intpntDualFeas"
: Dual feasibility measure reported by the interior-point optimizer. (For the interior-point optimizer this measure is not directly related to the original problem because a homogeneous model is employed.)

"intpntDualObj"
: Dual objective value reported by the interior-point optimizer.

"intpntFactorNumFlops"
: An estimate of the number of flops used in the factorization.

"intpntOptStatus"
: A measure of optimality of the solution. It should converge to $+1$ if the problem has a primal-dual optimal solution, and converge to -1 if the problem is (strictly) primal or dual infeasible. If the measure converges to another constant, or fails to settle, the problem is usually ill-posed.

"intpntOrderTime"
: Order time (in seconds).

"intpntPrimalFeas"
: Primal feasibility measure reported by the interior-point optimizer. (For the interior-point optimizer this measure is not directly related to the original problem because a homogeneous model is employed).

"intpntPrimalObj"
: Primal objective value reported by the interior-point optimizer.

"intpntTime"
: Time spent within the interior-point optimizer since its invocation.

"mioCliqueSeparationTime"
: Separation time for clique cuts.

"mioCmirSeparationTime"
: Separation time for CMIR cuts.

"mioConstructSolutionObj"
: If **MOSEK** has successfully constructed an integer feasible solution, then this item contains the optimal objective value corresponding to the feasible solution.

"mioDualBoundAfterPresolve"
: Value of the dual bound after presolve but before cut generation.

"mioGmiSeparationTime"
: Separation time for GMI cuts.

"mioImpliedBoundTime"
: Separation time for implied bound cuts.

"mioKnapsackCoverSeparationTime"
: Separation time for knapsack cover.

"mioObjAbsGap"
: Given the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the absolute gap defined by

$$|(\text{objective value of feasible solution}) - (\text{objective bound})|.$$

Otherwise it has the value -1.0.

"mioObjBound"
: The best known bound on the objective function. This value is undefined until at least one relaxation has been solved: To see if this is the case check that *"mioNumRelax"* is strictly positive.

"mioObjInt"
: The primal objective value corresponding to the best integer feasible solution. Please note that at least one integer feasible solution must have been located i.e. check *"mioNumIntSolutions"*.

"mioObjRelGap"
: Given that the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the relative gap defined by

$$\frac{|(\text{objective value of feasible solution}) - (\text{objective bound})|}{\max(\delta, |(\text{objective value of feasible solution})|)}.$$

where δ is given by the parameter *mioRelGapConst*. Otherwise it has the value -1.0.

"mioProbingTime"
: Total time for probing.

"mioRootCutgenTime"
: Total time for cut generation.

"mioRootOptimizerTime"
: Time spent in the optimizer while solving the root node relaxation

"mioRootPresolveTime"
: Time spent presolving the problem at the root node.

"mioTime"
: Time spent in the mixed-integer optimizer.

"mioUserObjCut"
: If the objective cut is used, then this information item has the value of the cut.

"optimizerTime"
: Total time spent in the optimizer since it was invoked.

"presolveEliTime"
: Total time spent in the eliminator since the presolve was invoked.

"presolveLindepTime"
: Total time spent in the linear dependency checker since the presolve was invoked.

"presolveTime"
: Total time (in seconds) spent in the presolve since it was invoked.

"primalRepairPenaltyObj"
: The optimal objective value of the penalty function.

"qcqoReformulateMaxPerturbation"
: Maximum absolute diagonal perturbation occurring during the QCQO reformulation.

"qcqoReformulateTime"
: Time spent with conic quadratic reformulation.

"qcqoReformulateWorstCholeskyColumnScaling"
: Worst Cholesky column scaling.

"qcqoReformulateWorstCholeskyDiagScaling"
: Worst Cholesky diagonal scaling.

"rdTime"
: Time spent reading the data file.

"simDualTime"
: Time spent in the dual simplex optimizer since invoking it.

"simFeas"
: Feasibility measure reported by the simplex optimizer.

"simObj"
: Objective value reported by the simplex optimizer.

"simPrimalTime"
: Time spent in the primal simplex optimizer since invoking it.

"simTime"
: Time spent in the simplex optimizer since invoking it.

"solBasDualObj"
: Dual objective value of the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasDviolcon"
: Maximal dual bound violation for x^c in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasDviolvar"
: Maximal dual bound violation for x^x in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasNrmBarx"
: Infinity norm of $\overline{X}$ in the basic solution.

"solBasNrmSlc"
: Infinity norm of s_l^c in the basic solution.

"solBasNrmSlx"
: Infinity norm of s_l^x in the basic solution.

"solBasNrmSuc"
: Infinity norm of s_u^c in the basic solution.

"solBasNrmSux"
: Infinity norm of s_u^X in the basic solution.

"solBasNrmXc"
: Infinity norm of x^c in the basic solution.

"solBasNrmXx"
: Infinity norm of x^x in the basic solution.

"solBasNrmY"
: Infinity norm of y in the basic solution.

"solBasPrimalObj"
: Primal objective value of the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasPviolcon"
: Maximal primal bound violation for x^c in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasPviolvar"
: Maximal primal bound violation for x^x in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solItgNrmBarx"
: Infinity norm of $\overline{X}$ in the integer solution.

"solItgNrmXc"
: Infinity norm of x^c in the integer solution.

"solItgNrmXx"
: Infinity norm of x^x in the integer solution.

"solItgPrimalObj"
: Primal objective value of the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolbarvar"
: Maximal primal bound violation for $\overline{X}$ in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolcon"
: Maximal primal bound violation for x^c in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolcones"
: Maximal primal violation for primal conic constraints in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolitg"
: Maximal violation for the integer constraints in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolvar"
: Maximal primal bound violation for x^x in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItrDualObj"
: Dual objective value of the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolbarvar"
: Maximal dual bound violation for $\overline{X}$ in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolcon"
: Maximal dual bound violation for x^c in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolcones"
: Maximal dual violation for dual conic constraints in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolvar"
: Maximal dual bound violation for x^x in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrNrmBars"
: Infinity norm of $\overline{S}$ in the interior-point solution.

"solItrNrmBarx"
: Infinity norm of $\overline{X}$ in the interior-point solution.

"solItrNrmSlc"
: Infinity norm of s_l^c in the interior-point solution.

"solItrNrmSlx"
: Infinity norm of s_l^x in the interior-point solution.

"solItrNrmSnx"
: Infinity norm of s_n^x in the interior-point solution.

"solItrNrmSuc"
: Infinity norm of s_u^c in the interior-point solution.

"solItrNrmSux"
: Infinity norm of s_u^X in the interior-point solution.

"solItrNrmXc"
: Infinity norm of x^c in the interior-point solution.

"solItrNrmXx"
: Infinity norm of x^x in the interior-point solution.

"solItrNrmY"
: Infinity norm of y in the interior-point solution.

"solItrPrimalObj"
: Primal objective value of the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolbarvar"
: Maximal primal bound violation for $\overline{X}$ in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolcon"
: Maximal primal bound violation for x^c in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolcones"
: Maximal primal violation for primal conic constraints in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolvar"
: Maximal primal bound violation for x^x in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"toConicTime"
: Time spent in the last to conic reformulation.

14.6.21 License feature

"pts"
: Base system.

"pton"
: Conic extension.

14.6.22 Long integer information items.

"biCleanDualDegIter"
: Number of dual degenerate clean iterations performed in the basis identification.

"biCleanDualIter"
: Number of dual clean iterations performed in the basis identification.

"biCleanPrimalDegIter"
: Number of primal degenerate clean iterations performed in the basis identification.

"biCleanPrimalIter"
: Number of primal clean iterations performed in the basis identification.

"biDualIter"
: Number of dual pivots performed in the basis identification.

"biPrimalIter"
: Number of primal pivots performed in the basis identification.

"intpntFactorNumNz"
: Number of non-zeros in factorization.

"mioAnz"
: Number of non-zero entries in the constraint matrix of the probelm to be solved by the mixed-integer optimizer.

"mioIntpntIter"
: Number of interior-point iterations performed by the mixed-integer optimizer.

"mioPresolvedAnz"
: Number of non-zero entries in the constraint matrix of the problem after the mixed-integer optimizer's presolve.

"mioSimplexIter"
: Number of simplex iterations performed by the mixed-integer optimizer.

"rdNumanz"
: Number of non-zeros in A that is read.

"rdNumqnz"
: Number of Q non-zeros.

14.6.23 Integer information items.

"anaProNumCon"
: Number of constraints in the problem.

"anaProNumConEq"
: Number of equality constraints.

"anaProNumConFr"
: Number of unbounded constraints.

"anaProNumConLo"
: Number of constraints with a lower bound and an infinite upper bound.

"anaProNumConRa"
: Number of constraints with finite lower and upper bounds.

"anaProNumConUp"
: Number of constraints with an upper bound and an infinite lower bound.

"anaProNumVar"
: Number of variables in the problem.

"anaProNumVarBin"
: Number of binary (0-1) variables.

"anaProNumVarCont"
: Number of continuous variables.

"anaProNumVarEq"
: Number of fixed variables.

"anaProNumVarFr"
: Number of free variables.

"anaProNumVarInt"
: Number of general integer variables.

"anaProNumVarLo"
: Number of variables with a lower bound and an infinite upper bound.

"anaProNumVarRa"
: Number of variables with finite lower and upper bounds.

"anaProNumVarUp"
Number of variables with an upper bound and an infinite lower bound.

"intpntFactorDimDense"
Dimension of the dense sub system in factorization.

"intpntIter"
Number of interior-point iterations since invoking the interior-point optimizer.

"intpntNumThreads"
Number of threads that the interior-point optimizer is using.

"intpntSolveDual"
Non-zero if the interior-point optimizer is solving the dual problem.

"mioAbsgapSatisfied"
Non-zero if absolute gap is within tolerances.

"mioCliqueTableSize"
Size of the clique table.

"mioConstructSolution"
This item informs if **MOSEK** constructed an initial integer feasible solution.

- -1: tried, but failed,
- 0: no partial solution supplied by the user,
- 1: constructed feasible solution.

"mioNodeDepth"
Depth of the last node solved.

"mioNumActiveNodes"
Number of active branch and bound nodes.

"mioNumBranch"
Number of branches performed during the optimization.

"mioNumCliqueCuts"
Number of clique cuts.

"mioNumCmirCuts"
Number of Complemented Mixed Integer Rounding (CMIR) cuts.

"mioNumGomoryCuts"
Number of Gomory cuts.

"mioNumImpliedBoundCuts"
Number of implied bound cuts.

"mioNumIntSolutions"
Number of integer feasible solutions that have been found.

"mioNumKnapsackCoverCuts"
Number of clique cuts.

"mioNumRelax"
Number of relaxations solved during the optimization.

"mioNumRepeatedPresolve"
Number of times presolve was repeated at root.

"mioNumbin"
Number of binary variables in the problem to be solved by the mixed-integer optimizer.

"mioNumbinconevar"
Number of binary cone variables in the problem to be solved by the mixed-integer optimizer.

"mioNumcon"
Number of constraints in the problem to be solved by the mixed-integer optimizer.

"mioNumcone"
Number of cones in the problem to be solved by the mixed-integer optimizer.

"mioNumconevar"
Number of cone variables in the problem to be solved by the mixed-integer optimizer.

"mioNumcont"
Number of continuous variables in the problem to be solved by the mixed-integer optimizer.

"mioNumcontconevar"
Number of continuous cone variables in the problem to be solved by the mixed-integer optimizer.

"mioNumdexpcones"
: Number of dual exponential cones in the problem to be solved by the mixed-integer optimizer.

"mioNumdpowcones"
: Number of dual power cones in the problem to be solved by the mixed-integer optimizer.

"mioNumint"
: Number of integer variables in the problem to be solved by the mixed-integer optimizer.

"mioNumintconevar"
: Number of integer cone variables in the problem to be solved by the mixed-integer optimizer.

"mioNumpexpcones"
: Number of primal exponential cones in the problem to be solved by the mixed-integer optimizer.

"mioNumppowcones"
: Number of primal power cones in the problem to be solved by the mixed-integer optimizer.

"mioNumqcones"
: Number of quadratic cones in the problem to be solved by the mixed-integer optimizer.

"mioNumrqcones"
: Number of rotated quadratic cones in the problem to be solved by the mixed-integer optimizer.

"mioNumvar"
: Number of variables in the problem to be solved by the mixed-integer optimizer.

"mioObjBoundDefined"
: Non-zero if a valid objective bound has been found, otherwise zero.

"mioPresolvedNumbin"
: Number of binary variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumbinconevar"
: Number of binary cone variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumcon"
: Number of constraints in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumcone"
: Number of cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumconevar"
: Number of cone variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumcont"
: Number of continuous variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumcontconevar"
: Number of continuous cone variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumdexpcones"
: Number of dual exponential cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumdpowcones"
: Number of dual power cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumint"
: Number of integer variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumintconevar"
: Number of integer cone variables in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumpexpcones"
: Number of primal exponential cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumppowcones"
: Number of primal power cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumqcones"
: Number of quadratic cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumrqcones"
: Number of rotated quadratic cones in the problem after the mixed-integer optimizer's presolve.

"mioPresolvedNumvar"
: Number of variables in the problem after the mixed-integer optimizer's presolve.

"mioRelgapSatisfied"
: Non-zero if relative gap is within tolerances.

"mioTotalNumCuts"
: Total number of cuts generated by the mixed-integer optimizer.

"mioUserObjCut"
: If it is non-zero, then the objective cut is used.

"optNumcon"
: Number of constraints in the problem solved when the optimizer is called.

"optNumvar"
: Number of variables in the problem solved when the optimizer is called

"optimizeResponse"
: The response code returned by optimize.

"purifyDualSuccess"
: Is nonzero if the dual solution is purified.

"purifyPrimalSuccess"
: Is nonzero if the primal solution is purified.

"rdNumbarvar"
: Number of symmetric variables read.

"rdNumcon"
: Number of constraints read.

"rdNumcone"
: Number of conic constraints read.

"rdNumintvar"
: Number of integer-constrained variables read.

"rdNumq"
: Number of nonempty Q matrices read.

"rdNumvar"
: Number of variables read.

"rdProtype"
: Problem type.

"simDualDegIter"
: The number of dual degenerate iterations.

"simDualHotstart"
: If 1 then the dual simplex algorithm is solving from an advanced basis.

"simDualHotstartLu"
: If 1 then a valid basis factorization of full rank was located and used by the dual simplex algorithm.

"simDualInfIter"
: The number of iterations taken with dual infeasibility.

"simDualIter"
: Number of dual simplex iterations during the last optimization.

"simNumcon"
: Number of constraints in the problem solved by the simplex optimizer.

"simNumvar"
: Number of variables in the problem solved by the simplex optimizer.

"simPrimalDegIter"
: The number of primal degenerate iterations.

"simPrimalHotstart"
: If 1 then the primal simplex algorithm is solving from an advanced basis.

"simPrimalHotstartLu"
: If 1 then a valid basis factorization of full rank was located and used by the primal simplex algorithm.

"simPrimalInfIter"
: The number of iterations taken with primal infeasibility.

"simPrimalIter"
: Number of primal simplex iterations during the last optimization.

"simSolveDual"
: Is non-zero if dual problem is solved.

"solBasProsta"
: Problem status of the basic solution. Updated after each optimization.

"solBasSolsta"
: Solution status of the basic solution. Updated after each optimization.

"solItgProsta"
: Problem status of the integer solution. Updated after each optimization.

"solItgSolsta"
: Solution status of the integer solution. Updated after each optimization.

"solItrProsta"
: Problem status of the interior-point solution. Updated after each optimization.

"solItrSolsta"
: Solution status of the interior-point solution. Updated after each optimization.

"stoNumARealloc"
: Number of times the storage for storing A has been changed. A large value may indicates that memory fragmentation may occur.

14.6.24 Information item types

"douType"
: Is a double information type.

"intType"
: Is an integer.

"lintType"
: Is a long integer.

14.6.25 Input/output modes

"read"
: The file is read-only.

"write"
: The file is write-only. If the file exists then it is truncated when it is opened. Otherwise it is created when it is opened.

"readwrite"
: The file is to read and write.

14.6.26 Specifies the branching direction.

"free"
: The mixed-integer optimizer decides which branch to choose.

"up"
: The mixed-integer optimizer always chooses the up branch first.

"down"
: The mixed-integer optimizer always chooses the down branch first.

"near"
: Branch in direction nearest to selected fractional variable.

"far"
: Branch in direction farthest from selected fractional variable.

"rootLp"
: Chose direction based on root lp value of selected variable.

"guided"
: Branch in direction of current incumbent.

"pseudocost"
: Branch based on the pseudocost of the variable.

14.6.27 Continuous mixed-integer solution type

"none"
: No interior-point or basic solution are reported when the mixed-integer optimizer is used.

"root"
: The reported interior-point and basic solutions are a solution to the root node problem when mixed-integer optimizer is used.

"itg"
: The reported interior-point and basic solutions are a solution to the problem with all integer variables fixed at the value they have in the integer solution. A solution is only reported in case the problem has a primal feasible solution.

"itgRel"

In case the problem is primal feasible then the reported interior-point and basic solutions are a solution to the problem with all integer variables fixed at the value they have in the integer solution. If the problem is primal infeasible, then the solution to the root node problem is reported.

14.6.28 Integer restrictions

"ignored"

The integer constraints are ignored and the problem is solved as a continuous problem.

"satisfied"

Integer restrictions should be satisfied.

14.6.29 Mixed-integer node selection types

"free"

The optimizer decides the node selection strategy.

"first"

The optimizer employs a depth first node selection strategy.

"best"

The optimizer employs a best bound node selection strategy.

"pseudo"

The optimizer employs selects the node based on a pseudo cost estimate.

14.6.30 MPS file format type

"strict"

It is assumed that the input file satisfies the MPS format strictly.

"relaxed"

It is assumed that the input file satisfies a slightly relaxed version of the MPS format.

"free"

It is assumed that the input file satisfies the free MPS format. This implies that spaces are not allowed in names. Otherwise the format is free.

"cplex"

The CPLEX compatible version of the MPS format is employed.

14.6.31 Objective sense types

"minimize"

The problem should be minimized.

"maximize"

The problem should be maximized.

14.6.32 On/off

"on"

Switch the option on.

"off"

Switch the option off.

14.6.33 Optimizer types

"conic"

The optimizer for problems having conic constraints.

"dualSimplex"

The dual simplex optimizer is used.

"free"

The optimizer is chosen automatically.

"freeSimplex"

One of the simplex optimizers is used.

"intpnt"
The interior-point optimizer is used.
"mixedInt"
The mixed-integer optimizer.
"primalSimplex"
The primal simplex optimizer is used.

14.6.34 Ordering strategies

"free"
The ordering method is chosen automatically.
"appminloc"
Approximate minimum local fill-in ordering is employed.
"experimental"
This option should not be used.
"tryGraphpar"
Always try the graph partitioning based ordering.
"forceGraphpar"
Always use the graph partitioning based ordering even if it is worse than the approximate minimum local fill ordering.
"none"
No ordering is used.

14.6.35 Presolve method.

"off"
The problem is not presolved before it is optimized.
"on"
The problem is presolved before it is optimized.
"free"
It is decided automatically whether to presolve before the problem is optimized.

14.6.36 Parameter type

"invalidType"
Not a valid parameter.
"douType"
Is a double parameter.
"intType"
Is an integer parameter.
"strType"
Is a string parameter.

14.6.37 Problem data items

"var"
Item is a variable.
"con"
Item is a constraint.
"cone"
Item is a cone.

14.6.38 Problem types

"lo"
The problem is a linear optimization problem.
"qo"
The problem is a quadratic optimization problem.

`"qcqo"`
: The problem is a quadratically constrained optimization problem.

`"conic"`
: A conic optimization.

`"mixed"`
: General nonlinear constraints and conic constraints. This combination can not be solved by **MOSEK**.

14.6.39 Problem status keys

`"unknown"`
: Unknown problem status.

`"primAndDualFeas"`
: The problem is primal and dual feasible.

`"primFeas"`
: The problem is primal feasible.

`"dualFeas"`
: The problem is dual feasible.

`"primInfeas"`
: The problem is primal infeasible.

`"dualInfeas"`
: The problem is dual infeasible.

`"primAndDualInfeas"`
: The problem is primal and dual infeasible.

`"illPosed"`
: The problem is ill-posed. For example, it may be primal and dual feasible but have a positive duality gap.

`"primInfeasOrUnbounded"`
: The problem is either primal infeasible or unbounded. This may occur for mixed-integer problems.

14.6.40 XML writer output mode

`"row"`
: Write in row order.

`"col"`
: Write in column order.

14.6.41 Response code type

`"ok"`
: The response code is OK.

`"wrn"`
: The response code is a warning.

`"trm"`
: The response code is an optimizer termination status.

`"err"`
: The response code is an error.

`"unk"`
: The response code does not belong to any class.

14.6.42 Scaling type

`"free"`
: The optimizer chooses the scaling heuristic.

`"none"`
: No scaling is performed.

`"moderate"`
: A conservative scaling is performed.

"aggressive"
A very aggressive scaling is performed.

14.6.43 Scaling method

"pow2"
Scales only with power of 2 leaving the mantissa untouched.
"free"
The optimizer chooses the scaling heuristic.

14.6.44 Sensitivity types

"basis"
Basis sensitivity analysis is performed.

14.6.45 Simplex selection strategy

"free"
The optimizer chooses the pricing strategy.
"full"
The optimizer uses full pricing.
"ase"
The optimizer uses approximate steepest-edge pricing.
"devex"
The optimizer uses devex steepest-edge pricing (or if it is not available an approximate steep-edge selection).
"se"
The optimizer uses steepest-edge selection (or if it is not available an approximate steep-edge selection).
"partial"
The optimizer uses a partial selection approach. The approach is usually beneficial if the number of variables is much larger than the number of constraints.

14.6.46 Solution items

"xc"
Solution for the constraints.
"xx"
Variable solution.
"y"
Lagrange multipliers for equations.
"slc"
Lagrange multipliers for lower bounds on the constraints.
"suc"
Lagrange multipliers for upper bounds on the constraints.
"slx"
Lagrange multipliers for lower bounds on the variables.
"sux"
Lagrange multipliers for upper bounds on the variables.
"snx"
Lagrange multipliers corresponding to the conic constraints on the variables.

14.6.47 Solution status keys

"unknown"
Status of the solution is unknown.
"optimal"
The solution is optimal.

"primFeas"
: The solution is primal feasible.

"dualFeas"
: The solution is dual feasible.

"primAndDualFeas"
: The solution is both primal and dual feasible.

"primInfeasCer"
: The solution is a certificate of primal infeasibility.

"dualInfeasCer"
: The solution is a certificate of dual infeasibility.

"primIllposedCer"
: The solution is a certificate that the primal problem is illposed.

"dualIllposedCer"
: The solution is a certificate that the dual problem is illposed.

"integerOptimal"
: The primal solution is integer optimal.

14.6.48 Solution types

"bas"
: The basic solution.

"itr"
: The interior solution.

"itg"
: The integer solution.

14.6.49 Solve primal or dual form

"free"
: The optimizer is free to solve either the primal or the dual problem.

"primal"
: The optimizer should solve the primal problem.

"dual"
: The optimizer should solve the dual problem.

14.6.50 Status keys

"unk"
: The status for the constraint or variable is unknown.

"bas"
: The constraint or variable is in the basis.

"supbas"
: The constraint or variable is super basic.

"low"
: The constraint or variable is at its lower bound.

"upr"
: The constraint or variable is at its upper bound.

"fix"
: The constraint or variable is fixed.

"inf"
: The constraint or variable is infeasible in the bounds.

14.6.51 Starting point types

"free"
: The starting point is chosen automatically.

"guess"
: The optimizer guesses a starting point.

"constant"
: The optimizer constructs a starting point by assigning a constant value to all primal and dual variables. This starting point is normally robust.

"satisfyBounds"
: The starting point is chosen to satisfy all the simple bounds on nonlinear variables. If this starting point is employed, then more care than usual should employed when choosing the bounds on the nonlinear variables. In particular very tight bounds should be avoided.

14.6.52 Stream types

"log"
: Log stream. Contains the aggregated contents of all other streams. This means that a message written to any other stream will also be written to this stream.

"msg"
: Message stream. Log information relating to performance and progress of the optimization is written to this stream.

"err"
: Error stream. Error messages are written to this stream.

"wrn"
: Warning stream. Warning messages are written to this stream.

14.6.53 Integer values

"maxStrLen"
: Maximum string length allowed in **MOSEK**.

"licenseBufferLength"
: The length of a license key buffer.

14.6.54 Variable types

"typeCont"
: Is a continuous variable.

"typeInt"
: Is an integer variable.

14.7 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*: An array did not have the required length, or two arrays were expected to have same length.
- *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.

14.7.1 Exception DimensionError

`mosek.fusion.DimensionError`

Thrown when a given object has the wrong number of dimensions, or they have not the right size.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.2 Exception DomainError

`mosek.fusion.DomainError`

Invalid domain.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.3 Exception ExpressionError

`mosek.fusion.ExpressionError`

Tried to construct an expression from invalid.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.4 Exception FatalError

`mosek.fusion.FatalError`

A fatal error has happened.

Implements `RuntimeException`

Members `RuntimeException.toString` – Return the exception message.

14.7.5 Exception FusionException

`mosek.fusion.FusionException`

Base class for all normal exceptions in fusion.

Implements `Exception`

Members *FusionException.toString* – Return the exception message.

Implemented by *SolutionError*

`FusionException.toString`

```
toString() -> str
```

Return the exception message.

Return (str)

14.7.6 Exception FusionRuntimeException

`mosek.fusion.FusionRuntimeException`

Base class for all run-time exceptions in fusion.

Implements `RuntimeException`

Members *FusionRuntimeException.toString* – Return the exception message.

Implemented by *ExpressionError*, *LengthError*, *DimensionError*, *SparseFormatError*, *SetDefinitionError*, *OptimizeError*, *ModelError*, *MatrixError*, *NameError*, *DomainError*, *IOError*, *SliceError*, *RangeError*, *ValueConversionError*, *ParameterError*, *IndexError*

`FusionRuntimeException.toString`

```
toString() -> str
```

Return the exception message.

Return (str)

14.7.7 Exception IOError

`mosek.fusion.IOError`

Error when reading or writing a stream, or opening a file.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.8 Exception IndexError

`mosek.fusion.IndexError`

Index out of bound, or a multi-dimensional index had wrong number of dimensions.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.9 Exception LengthError

`mosek.fusion.LengthError`

An array did not have the required length, or two arrays were expected to have same length.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.10 Exception MatrixError

mosek.fusion.MatrixError
: Thrown if data used in construction of a matrix contained inconsistencies or errors.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.11 Exception ModelError

mosek.fusion.ModelError
: Thrown when objects from different models were mixed.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.12 Exception NameError

mosek.fusion.NameError
: Name clash; tries to add a variable or constraint with a name that already exists.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.13 Exception OptimizeError

mosek.fusion.OptimizeError
: An error occurred during optimization.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.14 Exception ParameterError

mosek.fusion.ParameterError
: Tried to use an invalid parameter for a value that was invalid for a specific parameter.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.15 Exception RangeError

mosek.fusion.RangeError
: Invalid range specified

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.16 Exception SetDefinitionError

mosek.fusion.SetDefinitionError
: Invalid data for constructing set.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.17 Exception SliceError

mosek.fusion.SliceError
: Invalid slice definition, negative slice or slice index out of bounds.

  Implements *FusionRuntimeException*

  Members *FusionRuntimeException.toString* – Return the exception message.

14.7.18 Exception SolutionError

`mosek.fusion.SolutionError`

Requested a solution that was undefined or whose status was not acceptable.

Implements *FusionException*

Members *FusionException.toString* – Return the exception message.

14.7.19 Exception SparseFormatError

`mosek.fusion.SparseFormatError`

The given sparsity patters was invalid or specified an index that was out of bounds.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.7.20 Exception UnexpectedError

`mosek.fusion.UnexpectedError`

An unexpected error has happened. No specific excepion could have been risen.

Implements `RuntimeException`

Members `RuntimeException.toString` – Return the exception message.

14.7.21 Exception UnimplementedError

`mosek.fusion.UnimplementedError`

Called a stub. Functionality has not yet been implemented.

Implements `RuntimeException`

Members `RuntimeException.toString` – Return the exception message.

14.7.22 Exception ValueConversionError

`mosek.fusion.ValueConversionError`

Error casting or converting a value.

Implements *FusionRuntimeException*

Members *FusionRuntimeException.toString* – Return the exception message.

14.8 Class LinAlg

`mosek.LinAlg`

BLAS/LAPACK linear algebra routines.

Static members *LinAlg.axpy* – Computes vector addition and multiplication by a scalar.
LinAlg.dot – Computes the inner product of two vectors.
LinAlg.gemm – Performs a dense matrix multiplication.
LinAlg.gemv – Computes dense matrix times a dense vector product.
LinAlg.potrf – Computes a Cholesky factorization of a dense matrix.
LinAlg.syeig – Computes all eigenvalues of a symmetric dense matrix.
LinAlg.syevd – Computes all the eigenvalues and eigenvectors of a symmetric dense matrix, and thus its eigenvalue decomposition.
LinAlg.syrk – Performs a rank-k update of a symmetric matrix.

`LinAlg.axpy`

```
LinAlg.axpy(int n, float alpha, float[] x, float[] y)
```

Adds αx to y, i.e. performs the update

$$y := \alpha x + y.$$

Note that the result is stored overwriting y.

Parameters

- `n` (`int`) – Length of the vectors.
- `alpha` (`float`) – The scalar that multiplies x.
- `x` (`float`[]) – The x vector.
- `y` (`float`[]) – The y vector.

`LinAlg.dot`

```
LinAlg.dot(int n, float[] x, float[] y) -> float
```

Computes the inner product of two vectors x, y of length $n \geq 0$, i.e

$$x \cdot y = \sum_{i=1}^{n} x_i y_i.$$

Note that if $n = 0$, then the result of the operation is 0.

Parameters

- `n` (`int`) – Length of the vectors.
- `x` (`float`[]) – The x vector.
- `y` (`float`[]) – The y vector.

Return (`float`)

`LinAlg.gemm`

```
LinAlg.gemm(mosek.transpose transa, mosek.transpose transb, int m, int n, int k, float␣
↪alpha, float[] a, float[] b, float beta, float[] c)
```

Performs a matrix multiplication plus addition of dense matrices. Given A, B and C of compatible dimensions, this function computes

$$C := \alpha op(A) op(B) + \beta C$$

where α, β are two scalar values. The function $op(X)$ denotes X if transX is `NO`, or X^T if set to `YES`. The matrix C has m rows and n columns, and the other matrices must have compatible dimensions.

The result of this operation is stored in C.

Parameters

- `transa` (`transpose`) – Indicates if A should be transposed. See the Optimizer API documentation for the definition of these constants.
- `transb` (`transpose`) – Indicates if B should be transposed. See the Optimizer API documentation for the definition of these constants.
- `m` (`int`) – Indicates the number of rows of matrix C.
- `n` (`int`) – Indicates the number of columns of matrix C.
- `k` (`int`) – Specifies the common dimension along which $op(A)$ and $op(B)$ are multiplied. For example, if neither A nor B are transposed, then this is the number of columns in A and also the number of rows in B.

- `alpha` (`float`) – A scalar value multiplying the result of the matrix multiplication.
- `a` (`float`[]) – The pointer to the array storing matrix A in a column-major format.
- `b` (`float`[]) – The pointer to the array storing matrix B in a column-major format.
- `beta` (`float`) – A scalar value that multiplies C.
- `c` (`float`[]) – The pointer to the array storing matrix C in a column-major format.

`LinAlg.gemv`

```
LinAlg.gemv(mosek.transpose trans, int m, int n, float alpha, float[] a, float[] x, float␣
↪beta, float[] y)
```

Computes the multiplication of a scaled dense matrix times a dense vector, plus a scaled dense vector. Precisely, if `trans` is `NO` then the update is

$$y := \alpha Ax + \beta y,$$

and if `trans` is `YES` then

$$y := \alpha A^T x + \beta y,$$

where α, β are scalar values and A is a matrix with m rows and n columns.

Note that the result is stored overwriting y.

Parameters

- `trans` (`transpose`) – Indicates if A should be transposed. See the Optimizer API documentation for the definition of these constants.
- `m` (`int`) – Specifies the number of rows of the matrix A.
- `n` (`int`) – Specifies the number of columns of the matrix A.
- `alpha` (`float`) – A scalar value multiplying the matrix A.
- `a` (`float`[]) – A pointer to the array storing matrix A in a column-major format.
- `x` (`float`[]) – A pointer to the array storing the vector x.
- `beta` (`float`) – A scalar value multiplying the vector y.
- `y` (`float`[]) – A pointer to the array storing the vector y.

`LinAlg.potrf`

```
LinAlg.potrf(mosek.uplo uplo, int n, float[] a)
```

Computes a Cholesky factorization of a real symmetric positive definite dense matrix.

Parameters

- `uplo` (`uplo`) – Indicates whether the upper or lower triangular part of the matrix is used. See the Optimizer API documentation for the definition of these constants.
- `n` (`int`) – Specifies the dimension of the symmetric matrix.
- `a` (`float`[]) – A symmetric matrix stored in column-major order. Only the lower or the upper triangular part is used, accordingly with the `uplo` argument. It will contain the result on exit.

`LinAlg.syeig`

```
LinAlg.syeig(mosek.uplo uplo, int n, float[] a, float[] w)
```

Computes all eigenvalues of a real symmetric matrix A. Given a matrix $A \in \mathbb{R}^{n \times n}$ it returns a vector $w \in \mathbb{R}^n$ containing the eigenvalues of A.

Parameters

- `uplo` (`uplo`) – Indicates whether the upper or lower triangular part of the matrix is used. See the Optimizer API documentation for the definition of these constants.
- `n` (`int`) – Specifies the dimension of the symmetric matrix.
- `a` (`float`[]) – A symmetric matrix stored in column-major order. Only the lower or the upper triangular part is used, accordingly with the `uplo` argument. It will contain the result on exit.
- `w` (`float`[]) – Array of length at least `n` containing the eigenvalues of A.

`LinAlg.syevd`

```
LinAlg.syevd(mosek.uplo uplo, int n, float[] a, float[] w)
```

Computes all the eigenvalues and eigenvectors a real symmetric matrix. Given the input matrix $A \in \mathbb{R}^{n \times n}$, this function returns a vector $w \in \mathbb{R}^n$ containing the eigenvalues of A and it also computes the eigenvectors of A. Therefore, this function computes the eigenvalue decomposition of A as

$$A = UVU^T,$$

where $V = \mathbf{diag}(w)$ and U contains the eigenvectors of A.

Note that the matrix U overwrites the input data A.

Parameters

- `uplo` (`uplo`) – Indicates whether the upper or lower triangular part of the matrix is used. See the Optimizer API documentation for the definition of these constants.
- `n` (`int`) – Specifies the dimension of the symmetric matrix.
- `a` (`float`[]) – A symmetric matrix stored in column-major order. Only the lower or the upper triangular part is used, accordingly with the `uplo` argument. It will contain the result on exit.
- `w` (`float`[]) – Array of length at least `n` containing the eigenvalues of A.

`LinAlg.syrk`

```
LinAlg.syrk(mosek.uplo uplo, mosek.transpose trans, int n, int k, float alpha, float[] a,␣
↪float beta, float[] c)
```

Performs a symmetric rank-k update for a symmetric matrix.

Given a symmetric matrix $C \in \mathbb{R}^{n \times n}$, two scalars α, β and a matrix A of rank $k \leq n$, it computes either

$$C := \alpha A A^T + \beta C,$$

when `trans` is set to `NO` and $A \in \mathbb{R}^{n \times k}$, or

$$C := \alpha A^T A + \beta C,$$

when `trans` is set to `YES` and $A \in \mathbb{R}^{k \times n}$.

Only the part of C indicated by `uplo` is used and only that part is updated with the result.

Parameters

- `uplo` (`uplo`) – Indicates whether the upper or lower triangular part of C is used. See the Optimizer API documentation for the definition of these constants.
- `trans` (`transpose`) – Indicates if A should be transposed. See the Optimizer API documentation for the definition of these constants.
- `n` (`int`) – Specifies the order of C.
- `k` (`int`) – Indicates the number of rows or columns of A, depending on whether or not it is transposed, and its rank.
- `alpha` (`float`) – A scalar value multiplying the result of the matrix multiplication.
- `a` (`float`[]) – The pointer to the array storing matrix A in a column-major format.
- `beta` (`float`) – A scalar value that multiplies C.
- `c` (`float`[]) – The pointer to the array storing matrix C in a column-major format.

Chapter 15

Supported File Formats

MOSEK supports a range of problem and solution formats listed in Table 15.1 and Table 15.2. 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

Table 15.1: List of supported file formats for optimization problems. The column *Conic* refers to conic problems involving the quadratic, rotated quadratic, power or exponential cone. The last two columns indicate if the format supports solutions and optimizer parameters.

Format Type	Ext.	Binary/Text	LP	QO	Conic	SDP	Sol	Param
LP	lp	plain text	X	X				
MPS	mps	plain text	X	X	X			
OPF	opf	plain text	X	X	X		X	X
PTF	ptf	plain text	X	X	X	X	X	
CBF	cbf	plain text	X		X	X		
Task format	task	binary	X	X	X	X	X	X
Jtask format	jtask	text	X	X	X	X	X	X

Solution formats

Table 15.2: 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 and Zstandard compression. Problem files with extension `.gz` (for GZIP) and `.zst` (for Zstandard) 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.

15.1 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 of 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.

15.1.1 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 (`[ ]/2`) 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 per line, but **MOSEK** supports defining ranged constraints by using double-colon (`::`) instead of a single-colon (`:`) after the constraint name, i.e.

$$-5 \le x_1 + x_2 \le 5 \tag{15.1}$$

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 (15.1) may be written as

$$x_1 + x_2 - sl_1 = 0, \ -5 \le sl_1 \le 5.$$

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
```

15.1.2 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
```

15.1.3 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).

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* 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.
- *writeLpTermsPerLine* 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).

15.2 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]*.

15.2.1 MPS File Structure

The version of the MPS format supported by **MOSEK** allows specification of an optimization problem of the form

$$\begin{array}{lccccc} \text{maximize/minimize} & & c^T x + q_0(x) & & \\ l^c & \leq & Ax + q(x) & \leq & u^c, \\ l^x & \leq & x & \leq & u^x, \\ & & x \in \mathcal{K}, & & \\ & & x_{\mathcal{J}} \quad \text{integer}, & & \end{array} \tag{15.2}$$

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. The same applies to q_0.
- $\mathcal{K}$ is a convex cone.
- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer-constrained variables.
- c is the vector of objective coefficients.

An MPS file with one row and one column can be illustrated like this:

```
*        1         2         3         4         5         6
*23456789012345678901234567890123456789012345678901234567890
NAME          [name]
OBJSENSE
    [objsense]
OBJNAME       [objname]
ROWS
 ?  [cname1]
COLUMNS
    [vname1]  [cname1]  [value1]       [cname2]  [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

  ```
  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 Sec. 15.2.5 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.

`NAME` (optional)

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. `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 a 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 values `E`, `G`, `L`, or `N` with the following interpretation:

Constraint type	l_i^c	u_i^c
`E` (equal)	finite	$= l_i^c$
`G` (greater)	finite	∞
`L` (lower)	$-\infty$	finite
`N` (none)	$-\infty$	∞

In the MPS format the 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, unless something else was specified in the section `OBJNAME`.

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 -h constraint and v_1 denotes the value specified by `[value1]`, then the interpretation of v_1 is:

Constraint	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 $+$		$l_i^c + \vert v_1 \vert$
`L`	$-$ or $+$	$u_i^c - \vert v_1 \vert$	
`N`			

Another constraint bound can optionally be modified using `[cname2]` and `[value2]` the same way.

`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 terms belong. 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

$$\begin{array}{lccl} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* 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 `QSECTION`s please note that:

- Only one `QSECTION` is allowed for each constraint.
- The `QSECTION`s 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 stores all the nonzeros coefficients, without taking advantage of the symmetry of the Q matrix.
- QUADOBJ stores 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 appear 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 $1/2x^TQx$.

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 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
```

(continues on next page)

(continued from previous page)

```
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.

QCMATRIX (optional)

A QCMATRIX section allows to specify the quadratic part of a given constraint. 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 `QCMATRIX`s please note that:

- Only one `QCMATRIX` is allowed for each constraint.
- The `QCMATRIX`s 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.
- `QCMATRIX` does not exploit the symmetry of Q: an off-diagonal entry (i, j) should appear twice.

`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 for which the 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

Here v_1 is the value specified by `[value1]`.

`CSECTION` (optional)

The purpose of the `CSECTION` is to specify the conic constraint

$$x \in \mathcal{K}$$

in (15.2). 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:

$$\mathcal{K}_t = \mathbb{R}^{n^t}.$$

- Zero cone:

$$\mathcal{K}_t = \{0\} \subseteq \mathbb{R}^{n^t}. \tag{15.3}$$

- Quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1 \geq \sqrt{\sum_{j=2}^{n^t} x_j^2} \right\}. \tag{15.4}$$

- Rotated quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : 2x_1 x_2 \geq \sum_{j=3}^{n^t} x_j^2, \quad x_1, x_2 \geq 0 \right\}. \tag{15.5}$$

- Primal exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0 \right\}. \tag{15.6}$$

- Primal power cone (with parameter $0 < \alpha < 1$):

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{15.7}$$

- Dual exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0 \right\}. \tag{15.8}$$

- Dual power cone (with parameter $0 < \alpha < 1$):

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : \left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{15.9}$$

In general, membership in the $\mathbb{R}$ set is not specified. If a variable is not a member of any other cone then it is assumed to be a member of the $\mathbb{R}$ cone.

Next, let us study an example. Assume that the power cone

$$x_4^{1/3} x_5^{2/3} \geq |x_8|$$

and the rotated quadratic cone

$$2x_3 x_7 \geq x_1^2 + x_0^2, \quad x_3, x_7 \geq 0,$$

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     3e-1           PPOW
x4
x5
x8
CSECTION      koneb     0.0            RQUAD
x7
x3
x1
x0
```

In general, a CSECTION header has the format

```
CSECTION      [kname1]    [value1]     [ktype]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	Required	Description
[kname1]	15	8	Yes	Name of the cone
[value1]	25	12	No	Cone parameter
[ktype]	40		Yes	Type of the cone.

The possible cone type keys are:

[ktype]	Members	[value1]	Interpretation.
ZERO	≥ 0	unused	Zero cone (15.3).
QUAD	≥ 1	unused	Quadratic cone (15.4).
RQUAD	≥ 2	unused	Rotated quadratic cone (15.5).
PEXP	3	unused	Primal exponential cone (15.6).
PPOW	≥ 2	α	Primal power cone (15.7).
DEXP	3	unused	Dual exponential cone (15.8).
DPOW	≥ 2	α	Dual power cone (15.9).

A record in the CSECTION has the format

```
[vname1]
```

where the requirements for each field are

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	A valid variable name

A variable must occur in at most one CSECTION.

ENDATA

This keyword denotes the end of the MPS file.

15.2.2 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

- 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. MARKO001 and MARKO01, 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.

15.2.3 General Limitations

- An MPS file should be an ASCII file.

15.2.4 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.

15.2.5 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, a name must not contain any blanks.

15.3 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.

15.3.1 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 [/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]
```

15.3.2 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 subsections `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 subsections `b` (linear bounds on variables) and `cone` (cones).

[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]

Specifies a cone. A cone is defined as a sequence of variables which belong to a single unique cone. The supported cone types are:

- `quad`: a quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^2 \geq \sum_{i=2}^{n} x_i^2, \quad x_1 \geq 0.$$

- `rquad`: a rotated quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$2x_1x_2 \geq \sum_{i=3}^{n} x_i^2, \quad x_1, x_2 \geq 0.$$

- `pexp`: primal exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0.$$

- `ppow` with parameter $0 < \alpha < 1$: primal power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `dexp`: dual exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0.$$

- `dpow` with parameter $0 < \alpha < 1$: dual power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$\left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `zero`: zero cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1 = \cdots = x_n = 0$$

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 rquad] x,y,z,w  [/cone] # rotated quadratic cone
[cone ppow '3e-01' 'a'] x1, x2, x3 [/cone] # power cone with alpha=1/3 and name 'a'
[/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-valued.

`[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 is defined as follows:

```
[hint ITEM] value [/hint]
```

The hints recognized by **MOSEK** are:

- `numvar` (number of variables),
- `numcon` (number of linear/quadratic constraints),
- `numanz` (number of linear non-zeros in constraints),
- `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]
```

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.

15.3.3 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]+)?
```

15.3.4 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"
```

15.3.5 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]
```

15.3.6 Writing OPF Files from MOSEK

To write an OPF file add the `.opf` extension to the file name.

15.3.7 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 15.1.

Listing 15.1: 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

$$\begin{array}{lcl} \text{minimize} & & x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\ \text{subject to} & 1 \leq & x_1 + x_2 + x_3, \\ & & x \geq 0. \end{array}$$

This can be formulated in `opf` as shown below.

Listing 15.2: 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.
```

(continues on next page)

(continued from previous page)

```
   - 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:

$$\begin{array}{lccl} \text{minimize} & x_3 + x_4 + x_5 & & \\ \text{subject to} & x_0 + x_1 + 2x_2 & = & 1, \\ & x_0, x_1, x_2 & \geq & 0, \\ & x_3 \geq \sqrt{x_0^2 + x_1^2}, & & \\ & 2x_4 x_5 \geq x_2^2. & & \end{array}$$

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 15.3.

Listing 15.3: 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
```

(continues on next page)

(continued from previous page)

```
  [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 15.4.

Listing 15.4: 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]
```

15.4 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.

15.4.1 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:

$$\begin{array}{lll} \min / \max & g^{obj} & \\ & g_i \in \mathcal{K}_i, & i \in \mathcal{I}, \\ \text{s.t.} & G_i \in \mathcal{K}_i, & i \in \mathcal{I}^{PSD}, \\ & x_j \in \mathcal{K}_j, & j \in \mathcal{J}, \\ & \overline{X}_j \in \mathcal{K}_j, & j \in \mathcal{J}^{PSD}. \end{array} \tag{15.10}$$

- **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

$$\{x \in \mathbb{R}^n \mid x_j \geq 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Negative orthant** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j \leq 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Fixpoint zero** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j = 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R} \times \mathbb{R}^{n-1}, \ p^2 \geq x^T x, \ p \geq 0 \right\}, \text{ for } n \geq 2.$$

- **Rotated quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \begin{pmatrix} p \\ q \\ x \end{pmatrix} \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n-2}, \ 2pq \geq x^T x, \ p \geq 0, \ q \geq 0 \right\}, \text{ for } n \geq 3.$$

15.4.2 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.

15.4.3 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 15.3. 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 15.3.

- **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 15.3: 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

15.4.4 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 15.3), 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 15.3), 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

15.4.5 CBF Format Examples

Minimal Working Example

The conic optimization problem (15.11) , has three variables in a quadratic cone - first one is integer - and an affine expression in domain 0 (equality constraint).

$$\begin{array}{ll} \text{minimize} & 5.1\, x_0 \\ \text{subject to} & 6.2\, x_1 + 7.3\, x_2 - 8.4 \in \{0\} \\ & x \in \mathcal{Q}^3,\ x_0 \in \mathbb{Z}. \end{array} \tag{15.11}$$

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 (15.12), has a semidefinite cone, a quadratic cone over unordered subindices, and two equality constraints.

$$
\begin{array}{lrcl}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, X_1 \right\rangle + x_1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 & = & 1.0\,, \\
& \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, X_1 \right\rangle + x_0 + x_2 & = & 0.5\,, \\
& x_1 \geq \sqrt{x_0^2 + x_2^2}\,, & & \\
& X_1 \succeq \mathbf{0}\,. & &
\end{array}
\tag{15.12}
$$

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
```

(continues on next page)

(continued from previous page)

```
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.

$$
\begin{array}{lllll}
\text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 + x_2 + 1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X_1 \right\rangle - x_1 - x_2 & \geq & 0.0\,, \\
 & x_1 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq & \mathbf{0}\,, \\
 & X_1 \succeq \mathbf{0}\,. & &
\end{array}
\tag{15.13}
$$

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.
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
# 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 (15.14), is defined for a sequence of objectives such that hotstarting from one to the next might be advantages.

$$
\begin{array}{lrcl}
\text{maximize}_k & g_k^{obj} & & \\
\text{subject to} & 50\,x_0 + 31 & \leq & 250\,, \\
 & 3\,x_0 - 2x_1 & \geq & -4\,, \\
 & x \in \mathbb{R}_+^2, & &
\end{array}
\tag{15.14}
$$

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 15.5.

Listing 15.5: Problem (15.14) 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
```

(continues on next page)

(continued from previous page)

```
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
```

15.5 The PTF Format

The PTF format is a new human-readable, natural text format. Its features and structure are similar to the *OPF* format, with the difference that the PTF format **does** support semidefinite terms.

15.5.1 The overall format

The format is indentation based, where each section is started by a head line and followed by a section body with deeper indentation that the head line. For example:

```
Header line
   Body line 1
   Body line 1
   Body line 1
```

Section can also be nested:

```
Header line A
   Body line in A
   Header line A.1
      Body line in A.1
      Body line in A.1
   Body line in A
```

The indentation of blank lines is ignored, so a subsection can contain a blank line with no indentation. The character # defines a line comment and anything between the # character and the end of the line is ignored.

In a PTF file, the first section must be a `Task` section. The order of the remaining section is arbitrary, and sections may occur multiple times or not at all.

MOSEK will ignore any top-level section it does not recognize.

Names

In the description of the format we use following definitions for name strings:

```
NAME: PLAIN_NAME | QUOTED_NAME
PLAIN_NAME: [a-zA-Z_] [a-zA-Z0-9_-.!|]
QUOTED_NAME: "'" ( [^'\\\r\n] | "\\" ( [\\rn] | "x" [0-9a-fA-F] [0-9a-fA-F] ) )* "'"
```

Expressions

An expression is a sum of terms. A term is either a linear term (a coefficient and a variable name, where the coefficient can be left out if it is 1.0), or a matrix inner product.

An expression:

```
EXPR: EMPTY | [+-]? TERM ( [+-] TERM )*
TERM: LINEAR_TERM | MATRIX_TERM
```

A linear term

```
LINEAR_TERM: FLOAT? NAME
```

A matrix term

```
MATRIX_TERM: "<" FLOAT? NAME ( [+-] FLOAT? NAME)* ";" NAME ">"
```

Here the right-hand name is the name of a (semidefinite) matrix variable, and the left-hand side is a sum of symmetric matrixes. The actual matrixes are defined in a separate section.

Expressions can span multiple lines by giving subsequent lines a deeper indentation.

For example following two section are equivalent:

```
# Everything on one line:
x1 + x2 + x3 + x4

# Split into multiple lines:
x1
  + x2
  + x3
  + x4
```

15.5.2 `Task` section

The first section of the file must be a `Task`. The text in this section is not used and may contain comments, or meta-information from the writer or about the content.

Format:

```
Task NAME
   Anything goes here...
```

`NAME` is a the task name.

15.5.3 Objective section

The `Objective` section defines the objective name, sense and function. The format:

```
"Objective" NAME?
   ( "Minimize" | "Maximize" ) EXPR
```

For example:

```
Objective 'obj'
   Minimize x1 + 0.2 x2 + < M1 ; X1 >
```

15.5.4 Constraints section

The constraints section defines a series of constraints. A constraint defines a term $A \cdot x + b \in K$. For linear constraints `A` is just one row, while for conic constraints it can be multiple rows. If a constraint spans multiple rows these can either be written inline separated by semi-colons, or each expression in a separete sub-section.

Simple linear constraints:

```
"Constraints"
  NAME? "[" [-+] (FLOAT | "Inf") (";" [-+] (FLOAT | "Inf") )? "]" EXPR
```

If the brackets contain two values, they are used as upper and lower bounds. It they contain one value the constraint is an equality.

For example:

```
Constraints
  'c1' [0;10] x1 + x2 + x3
  [0] x1 + x2 + x3
```

Constraint blocks put the expression either in a subsection or inline. The cone type (domain) is written in the brackets, and **MOSEK** currently supports following types:

- `SOC(N)` Second order cone of dimension `N`
- `RSOC(N)` Rotated second order cone of dimension `N`
- `PSD(N)` Symmetric positive semidefinite cone of dimension `N`. This contains `N*(N+1)/2` elements.
- `PEXP` Primal exponential cone of dimension 3
- `DEXP` Dual exponential cone of dimension 3
- `PPOW(N,P)` Primal power cone of dimension `N` with parameter `P`
- `DPOW(N,P)` Dual power cone of dimension `N` with parameter `P`
- `ZERO(N)` The zero-cone of dimension `N`.

```
"Constraints"
  NAME? "[" DOMAIN "]" EXPR_LIST
```

For example:

```
Constraints
   'K1' [SOC(3)] x1 + x2 ; x2 + x3 ; x3 + x1
   'K2' [RSOC(3)]
      x1 + x2
      x2 + x3
      x3 + x1
```

15.5.5 Variables section

Any variable used in an expression must be defined in a variable section. The variable section defines each variable domain.

```
"Variables"
   NAME "[" [-+] (FLOAT | "Inf") (";" [-+] (FLOAT | "Inf") )? "]"
   NAME "[" DOMAIN "]" NAMES

   For example, a linear variable
```

```
Variables
   x1 [0;Inf]
```

As with constraints, members of a conic domain can be listed either inline or in a subsection:

```
Variables
   k1 [SOC(3)] x1 ; x2 ; x3
   k2 [RSOC(3)]
      x1
      x2
      x3
```

15.5.6 Integer section

This section contains a list of variables that are integral. For example:

```
Integer
   x1 x2 x3
```

15.5.7 SymmetricMatrixes section

This section defines the symmetric matrixes used for matrix coefficients in matrix inner product terms. The section lists named matrixes, each with a size and a number of non-zeros. Only non-zeros in the lower triangular part should be defined.

```
"SymmetricMatrixes"
   NAME "SYMMAT" "(" INT ")"  ( "(" INT "," INT "," FLOAT ")" )*
   ...
```

For example:

```
SymmetricMatrixes
   M1 SYMMAT(3) (0,0,1.0) (1,1,2.0) (2,1,0.5)
   M2 SYMMAT(3)
      (0,0,1.0)
      (1,1,2.0)
      (2,1,0.5)
```

15.5.8 Solutions section

Each subsection defines a solution. A solution defines for each constraint and for each variable exactly one primal value and either one (for conic domains) or two (for linear domains) dual values. The values follow the same logic as in the **MOSEK** C API. A primal and a dual solution status defines the meaning of the values primal and dual (solution, certificate, unknown, etc.)

The format is this:

```
"Solutions"
   "Solution" WHICHSOL
      "ProblemStatus" PROSTA PROSTA?
   "SolutionStatus" SOLSTA SOLSTA?
```

(continues on next page)

(continued from previous page)

```
"Objective" FLOAT FLOAT
"Variables"
   # Linear variable status: level, slx, sux
   NAME "[" STATUS "]" FLOAT (FLOAT FLOAT)?
   # Conic variable status: level, snx
   NAME
      "[" STATUS "]" FLOAT FLOAT?
      ...
"Constraints"
   # Linear variable status: level, slx, sux
   NAME "[" STATUS "]" FLOAT (FLOAT FLOAT)?
   # Conic variable status: level, snx
   NAME
      "[" STATUS "]" FLOAT FLOAT?
      ...
```

Following values for `WHICHSOL` are supported:

- `interior` Interior solution, the result of an interior-point solver.
- `basic` Basic solution, as produced by a simplex solver.
- `integer` Integer solution, the solution to a mixed-integer problem. This does not define a dual solution.

Following values for `PROSTA` are supported:

- `unknown` The problem status is unknown
- `feasible` The problem has been proven feasible
- `infeasible` The problem has been proven infeasible
- `illposed` The problem has been proved to be ill posed
- `infeasible_or_unbounded` The problem is infeasible or unbounded

Following values for `SOLSTA` are supported:

- `unknown` The solution status is unknown
- `feasible` The solution is feasible
- `optimal` The solution is optimal
- `infeas_cert` The solution is a certificate of infeasibility
- `illposed_cert` The solution is a certificate of illposedness

Following values for `STATUS` are supported:

- `unknown` The value is unknown
- `super_basic` The value is super basic
- `at_lower` The value is basic and at its lower bound
- `at_upper` The value is basic and at its upper bound
- `fixed` The value is basic fixed
- `infinite` The value is at infinity

15.6 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, 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:

- Status of a solution read from a file will *always* be unknown.
- Parameter settings in a task file *always override* any parameters set on the command line or in a parameter file.

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.

15.7 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`

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.

15.7.1 *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**.

15.7.2 *jsol* format

It stores a problem solution. The *jsol* format contains all solutions and information items.

15.7.3 A *jtask* example

In Listing 15.6 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 15.6: 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"
            ],
```


(continues on next page)

(continued from previous page)


```
        "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,
```


(continues on next page)

(continued from previous page)


```
                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,
```


(continues on next page)

(continued from previous page)

```
                "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",
```

(continues on next page)

(continued from previous page)


```
            "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",
```


(continues on next page)

(continued from previous page)


```
        "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,
```


(continues on next page)

(continued from previous page)

```
            "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"
        }
    }
}
```

15.8 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

$$l_i^c \leq \sum_{j=1}^{n} a_{ij} x_j \leq u_i^c, \tag{15.15}$$

  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 15.4 the possible values of the status keys and their interpretation are shown.

Table 15.4: 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 (15.15).)
  - `UPPER LIMIT`: the quantity u_i^c (see (15.15).)
  - `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 15.7 we show the solution file for the lo1.opf problem.

Listing 15.7: 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     -2.49999999741653e+00
1          c2             SB 5.33333333049187e+01     1.50000000e+01     NONE               2.
↪09159033069640e-10      -0.00000000000000e+00
2          c3             UL 2.49999999842049e+01     NONE               2.50000000e+01     -0.
↪00000000000000e+00     -3.33333332895108e-01

VARIABLES
INDEX      NAME           AT ACTIVITY                 LOWER LIMIT        UPPER LIMIT        ␣
↪DUAL LOWER                DUAL UPPER
0          x1             LL 1.67020427038537e-09     0.00000000e+00     NONE               -4.
↪49999999528054e+00     -0.00000000000000e+00
1          x2             LL 2.93510446211883e-09     0.00000000e+00     1.00000000e+01     -2.
↪16666666494915e+00     6.20868657679896e-10
2          x3             SB 1.49999999899424e+01     0.00000000e+00     NONE               -8.
↪79123177245553e-10     -0.00000000000000e+00
3          x4             SB 8.33333332273115e+00     0.00000000e+00     NONE               -1.
↪69795978848200e-09     -0.00000000000000e+00
```

Chapter 16

List of examples

List of examples shipped in the distribution of Fusion API for Python:

Table 16.1: List of distributed examples

File	Description
TrafficNetworkModel.py	Demonstrates a traffic network problem as a conic quadratic problem (CQO)
alan.py	A portfolio choice model alan.gms from the GAMS model library
baker.py	A small bakery revenue maximization linear problem
breaksolver.py	Shows how to break a long-running task
callback.py	An example of data/progress callback
ceo1.py	A simple conic exponential problem
cqo1.py	A simple conic quadratic problem
diet.py	Solving Stigler's Nutrition model diet from the GAMS model library
duality.py	Shows how to access the dual solution
facility_location.py	Demonstrates a small one-facility location problem (CQO)
gp1.py	A simple geometric program (GP) in conic form
lo1.py	A simple linear problem
logistic.py	Implements logistic regression and simple log-sum-exp (CEO)
lownerjohn_ellipsoid.py	Computes the Lowner-John inner and outer ellipsoidal approximations of a polytope (SDO, Power Cone)
lpt.py	Demonstrates how to solve the multi-processor scheduling problem and input an integer feasible point (MIP)
mico1.py	A simple mixed-integer conic problem
milo1.py	A simple mixed-integer linear problem
mioinitsol.py	A simple mixed-integer linear problem with an initial guess
modelLib.py	Library of implementations of basic functions
nearestcorr.py	Solves the nearest correlation matrix problem (SDO, CQO)
parameters.py	Shows how to set optimizer parameters and read information items
portfolio_1_basic.py	Portfolio optimization - basic Markowitz model
portfolio_2_frontier.py	Portfolio optimization - efficient frontier
portfolio_3_impact.py	Portfolio optimization - market impact costs
portfolio_4_transcost.py	Portfolio optimization - transaction costs
portfolio_5_card.py	Portfolio optimization - cardinality constraints
pow1.py	A simple power cone problem
primal_svm.py	Implements a simple soft-margin Support Vector Machine (CQO)

Continued on next page

Table 16.1 – continued from previous page

File	Description
qcqp_sdo_relaxation.py	Demonstrate how to use SDP to solve convex relaxation of a mixed-integer QCQO problem
reoptimization.py	Demonstrate how to modify and re-optimize a linear problem
response.py	Demonstrates proper response handling
sdo1.py	A simple semidefinite optimization problem
sospoly.py	Models the cone of nonnegative polynomials and nonnegative trigonometric polynomials using Nesterov's framework
sudoku.py	A SUDOKU solver (MIP)
total_variation.py	Demonstrates how to solve a total variation problem (CQO)
tsp.py	Solves a simple Travelling Salesman Problem and shows how to add constraints to a model and re-optimize (MIP)

Additional examples can be found on the **MOSEK** website and in other **MOSEK** publications.

Chapter 17

Interface changes

The section shows interface-specific changes to the **MOSEK** Fusion API for Python in version 9.0. See the release notes for general changes and new features of the **MOSEK** Optimization Suite.

17.1 Backwards compatibility

There is a number of small changes in the *Fusion* API. Most of them should not affect standard applications of *Fusion* and very few should cause compilation errors.

- **Parameters.** Users who set parameters to tune the performance and numerical properties of the solver (termination criteria, tolerances, solving primal or dual, presolve etc.) are recommended to reevaluate such tuning. It may be that other, or default, parameter settings will be more beneficial in the current version. The hints in Sec. 9 may be useful for some cases.
- Remove all `Near` problem and solution statuses i.e. `SolutionStatus.NearOptimal`, `SolutionStatus.NearCertificate`, `AccSolutionStatus.NearOptimal`, etc. See Sec. 13.3.3.
- Some algebraic operators are more strict when it comes to exactly matching shapes, especially regarding shapes such as $1 \times n$, $n \times 1$ and n. The same applies to matching variable/expression and domain shapes. In some cases it may be necessary to explicitly reshape using the method *`Variable.reshape`* or *`LinearDomain.withShape`*.
- All shapes are specified with ordinary arrays instead of objects of class `Set`. The static methods of the class *`Set`* produce arrays as shape specifications.
- Replace `shape()` with `getShape()` and `size()` with `getSize()` in most places.
- Remove the option in *`Expr.sum`* to sum over a range of dimensions in a multidimensional expression.
- Remove the method `Constraint.add` and introduce *`Constraint.update`* (see Sec. 7.9).
- Rename `QConeDomain` to `ConeDomain`.
- *`Variable`* now extends *`Expression`*, so the number of method prototypes is reduced in some cases. A variable can be used everywhere an expression can.
- Expressions are evaluated lazily only when used in a constraint.
- **Semidefinite variables.** The preferred ways to declare semidefinite variables are:

```
M.variable(Domain.inPSDCone(3))         # A 3x3 PSD variable
M.variable(Domain.inPSDCone(3, 100))    # One hundred 3x3 PSD variables
                                        # arranged in shape [100, 3, 3]
```

17.2 Parameters

Added

- *intpntOrderGpNumSeeds*
- *logLocalInfo*
- *mioConicOuterApproximation*
- *mioFeaspumpLevel*
- *mioMaxNumRootCutRounds*
- *mioPropagateObjectiveConstraint*
- *mioSeed*
- *presolveMaxNumPass*
- *simSeed*

Removed

- dataTolAij
- intpntNlMeritBal
- intpntNlTolDfeas
- intpntNlTolMuRed
- intpntNlTolNearRel
- intpntNlTolPfeas
- intpntNlTolRelGap
- intpntNlTolRelStep
- mioDisableTermTime
- mioNearTolAbsGap
- mioNearTolRelGap
- mioConstructSol
- mioMtUserCb
- opfMaxTermsPerLine
- readDataCompressed
- readDataFormat
- writeDataCompressed
- writeDataFormat

17.3 Constants

Added

Removed

- `xml`
- `mioHeuristicTime`
- `mioOptimizerTime`
- `mioConstructNumRoundings`
- `mioInitialSolution`
- `mioNearAbsgapSatisfied`
- `mioNearRelgapSatisfied`
- `mioSimMaxiterSetbacks`
- `hybrid`
- `worst`
- `geco`
- `nearDualFeas`
- `nearPrimAndDualFeas`
- `nearPrimFeas`
- `optimalPartition`
- `nearDualFeas`
- `nearDualInfeasCer`
- `nearIntegerOptimal`
- `nearOptimal`
- `nearPrimAndDualFeas`
- `nearPrimFeas`
- `nearPrimInfeasCer`

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.

[And13] Erling D. Andersen. On formulating quadratic functions in optimization models. Technical Report TR-1-2013, MOSEK ApS, 2013. Last revised 23-feb-2016. URL: http://docs.mosek.com/whitepapers/qmodel.pdf.

[BKVH07] S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi. A Tutorial on Geometric Programming. *Optimization and Engineering*, 8(1):67–127, 2007. Available at http://www.stanford.edu/~boyd/gp_tutorial.html.

[Chv83] V. Chvátal. *Linear programming.* W.H. Freeman and Company, 1983.

[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.

Symbol Index

Classes

Enumerations

Exceptions

Parameters

Response codes

Index

T

U

V