



MOSEK Fusion API for Python
Release 9.3.21

MOSEK ApS

08 August 2022

Contents

1	Introduction	1
1.1	Why the Fusion API for Python?	2
2	Contact Information	3
3	License Agreement	4
4	Installation	8
4.1	Anaconda	8
4.2	PIP and Wheels	8
4.3	PyPy	9
4.4	Manual installation	9
4.5	Testing the Installation	9
4.6	Troubleshooting	10
5	Design Overview	11
6	Conic Modeling	13
6.1	The model	13
6.2	Variables	14
6.3	Linear algebra	14
6.4	Constraints and objective	15
6.5	Matrices	16
6.6	Parameters	17
6.7	Stacking and views	17
6.8	Vectorization	18
6.9	Reoptimization	19
7	Optimization Tutorials	20
7.1	Linear Optimization	20
7.2	Conic Quadratic Optimization	22
7.3	Power Cone Optimization	25
7.4	Conic Exponential Optimization	27
7.5	Semidefinite Optimization	29
7.6	Integer Optimization	33
7.7	Geometric Programming	36
7.8	Library of basic functions	38
7.9	Model Parametrization and Reoptimization	44
7.10	Problem Modification and Reoptimization	47
8	Solver Interaction Tutorials	51
8.1	Accessing the solution	51
8.2	Errors and exceptions	54
8.3	Input/Output	56
8.4	Setting solver parameters	57
8.5	Retrieving information items	58
8.6	Stopping the solver	59

8.7	Progress and data callback	60
8.8	Optimizer API Task	63
8.9	MOSEK OptServer	63
9	Debugging Tutorials	65
9.1	Understanding optimizer log	65
9.2	Addressing numerical issues	70
9.3	Debugging infeasibility	72
9.4	Python Console	76
10	Technical guidelines	79
10.1	Limitations	79
10.2	Memory management and garbage collection	79
10.3	Names	80
10.4	Multithreading	81
10.5	Efficiency	81
10.6	The license system	82
10.7	Deployment	83
11	Case Studies	84
11.1	Portfolio Optimization	85
11.2	Primal Support-Vector Machine (SVM)	96
11.3	2D Total Variation	101
11.4	Multiprocessor Scheduling	104
11.5	Logistic regression	108
11.6	Inner and outer Löwner-John Ellipsoids	110
11.7	SUDOKU	113
11.8	Travelling Salesman Problem (TSP)	117
11.9	Nearest Correlation Matrix Problem	121
11.10	Semidefinite Relaxation of MIQCQO Problems	125
12	Problem Formulation and Solutions	128
12.1	Linear Optimization	128
12.2	Conic Optimization	131
12.3	Semidefinite Optimization	135
13	Optimizers	137
13.1	Presolve	137
13.2	Linear Optimization	139
13.3	Conic Optimization - Interior-point optimizer	146
13.4	The Optimizer for Mixed-integer Problems	150
14	<i>Fusion</i> API Reference	154
14.1	<i>Fusion</i> API conventions	154
14.2	Class list	155
14.3	Parameters grouped by topic	235
14.4	Parameters (alphabetical list sorted by type)	243
14.5	Enumerations	269
14.6	Constants	270
14.7	Exceptions	293
14.8	Class LinAlg	297
15	Supported File Formats	302
15.1	The LP File Format	303
15.2	The MPS File Format	308
15.3	The OPF Format	320
15.4	The CBF Format	330
15.5	The PTF Format	344
15.6	The Task Format	349

15.7	The JSON Format	349
15.8	The Solution File Format	357
16	List of examples	360
17	Interface changes	362
17.1	Backwards compatibility	362
17.2	Parameters	363
17.3	Constants	364
	Bibliography	365
	Symbol Index	366
	Index	369

Chapter 1

Introduction

The **MOSEK** Optimization Suite 9.3.21 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. Moreover, parametrization makes it possible to construct a *Fusion* model once and then solve it repeatedly for different inputs with almost no overhead.

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
Linkedin	https://www.linkedin.com/company/mosek-aps
Youtube	https://www.youtube.com/channel/UCvIyectEVLp31NXeD5mIbEw

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/9.3/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.

```
/*
 *
 * 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
```

(continues on next page)

(continued from previous page)

(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.

mimalloc

MOSEK includes the *mimalloc* memory allocator library from [github/mimalloc](https://github.com/mimalloc). The license agreement for *mimalloc* is shown in [Listing 3.4](#).

Listing 3.4: *mimalloc* license.

MIT License

Copyright (c) 2019 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

blis

MOSEK includes the *blis* implementation of BLAS library from [github/blis](https://github.com/blis). The license agreement for *blis* is shown in [Listing 3.5](#).

Listing 3.5: *blis* license.

Copyright (C) 2018, The University of Texas at Austin
Copyright (C) 2016, Hewlett Packard Enterprise Development LP
Copyright (C) 2018 - 2019, Advanced Micro Devices, Inc.

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(s) of the copyright holder(s) nor the names of its contributors may be used to endorse or promote products derived

(continues on next page)

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.

OpenBLAS

MOSEK includes the *OpenBLAS* implementation of BLAS library from [github/OpenBLAS](https://github.com/OpenBLAS). The license agreement for *OpenBLAS* is shown in [Listing 3.6](#).

Listing 3.6: *openblas* license.

Copyright (c) 2011-2014, The OpenBLAS Project
All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.
3. Neither the name of the OpenBLAS project 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 OWNER 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:

Table 4.1: Supported Python versions.

Platform	Python	PyPy
Linux 64 bit x86	2.7, 3.6, 3.7, 3.8, 3.9, 3.10	2.7
Mac OS 64 bit	2.7, 3.6, 3.7, 3.8, 3.9, 3.10	2.7
Windows 64 bit	2.7, 3.6, 3.7, 3.8, 3.9, 3.10	2.7
Windows 32 bit	2.7, 3.6, 3.7	2.7
Linux ARM64	3.7, 3.8, 3.9	

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 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 4.2.

Table 4.2: Relevant files for the Fusion API for Python.

Relative Path	Description	Label
<code><MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/python/2</code>	Python 2 install	<code><PYTHON2DIR></code>
<code><MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/python/3</code>	Python 3 install	<code><PYTHON3DIR></code>
<code><MSKHOME>/mosek/9.3/tools/examples/fusion/python</code>	Examples	<code><EXDIR></code>
<code><MSKHOME>/mosek/9.3/tools/examples/fusion/data</code>	Additional data	<code><MISCDIR></code>

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`, `osx64x86` or `linuxaarch64`.

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
```

4.6 Troubleshooting

error: could not create 'build': Access is denied

If an attempt to install the Python interface results in an error such as

```
error: could not create 'build': Access is denied
```

then you have no write permissions to the folder where **MOSEK** is installed. This can happen for example if the package was installed by an administrator, and a user is trying to set up the Python interface. One solution is to install **MOSEK** in another location. Another solution is to specify the location of the build folder in a place the user can write to, for example:

```
python setup.py build --build-base=SOME_FOLDER install --user
```

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:

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

where μ, G are input data and α is an input parameter whose value we want to change between many optimizations. Its representation in *Fusion* is a direct translation of the mathematical model and could look as follows:

```
x = M.variable(n)
gamma = M.variable()
alpha = M.parameter()

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

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

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

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

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. They 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
<i>Expr.add</i>	Element-wise addition of two matrices
<i>Expr.sub</i>	Element-wise subtraction of two matrices
<i>Expr.mul</i>	Matrix or matrix-scalar multiplication
<i>Expr.neg</i>	Sign inversion
<i>Expr.outer</i>	Vector outer-product
<i>Expr.dot</i>	Dot product
<i>Expr.sum</i>	Sum over a given dimension
<i>Expr.mulElm</i>	Element-wise multiplication
<i>Expr.mulDiag</i>	Sum over the diagonal of a matrix which is the result of a matrix multiplication
<i>Expr.constTerm</i>	Return a <i>constant term</i>

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

$$\begin{array}{rcl} x_1 & + & 2x_2 & = & 0 \\ & & + & x_2 & + & x_3 & = & 0 \\ x_1 & & & = & 0 \end{array} \quad (6.2)$$

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. 6.8](#).

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

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 Parameters

A parameter (*Parameter*) is a placeholder for a constant whose value should be specified before the model is optimized. Parameters can have arbitrary shapes, just like variables, and can be used in any place where using a constant, array or matrix of the same shape would be suitable. That means parameters behave like expressions under additive operations and stacking, and can additionally be used in some multiplicative operations where the result is affine in the optimization variables.

For example, we can create a parametrized constraint

$$p^T x + q \leq 0,$$

where $x \in \mathbb{R}^4$, as follows:

```

x = M.variable('x', 4)                # Variable

p = M.parameter('p', 4)                # Parameter of shape [ 4 ]
q = M.parameter()                      # Scalar parameter

M.constraint(Expr.add(Expr.dot(p, x), q), Domain.lessThan(0.0))

```

Later in the code we can initialize the parameters with actual values. For example

```

p.setValue([1,2,3,4])
q.setValue(5)

```

will make the previously defined constraint evaluate to

$$x_1 + 2x_2 + 3x_3 + 4x_4 + 5 \leq 0.$$

The values of parameters can be changed between optimizations. Therefore one parametrized model with fixed structure can be used to solve many instances of the same optimization problem with varying input data.

6.7 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. 6.8](#),
- 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.

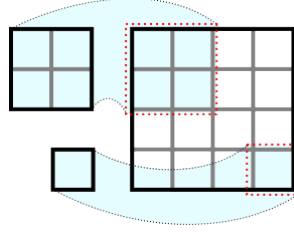


Fig. 6.1: Two dimensional slicing.

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

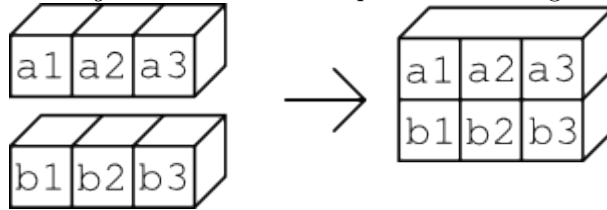
```
A1 = Ax.slice([0,0],[2,2])
A2 = Ax.index([3,3])
```

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 is *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.

6.8 Vectorization

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

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

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

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

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.

6.9 Reoptimization

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

- Set/change values of parameters (`Parameter.setValue`). This is the recommended way to reoptimize multiple models identical structure and varying (parts of) input data. For simplicity, suppose we want to minimize $f(x) = \gamma x + \beta y$, for varying choices of $\gamma > 0$. Then we could write:

```

gammaValues = [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.))
    gamma = M.parameter('gamma')

    M.objective( ObjectiveSense.Minimize, Expr.add(Expr.mul(gamma, x), Expr.
→mul(beta, y)) )

    for g in gammaValues:
        gamma.setValue(g)
        M.solve()

```

- 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` or parameters with `Model.parameter`.
- Replace the objective with a completely new one (`Model.objective`).
- 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. 7.10](#) for more reoptimization examples.

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

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

and the bounds

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

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

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

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

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

- l^c and u^c — 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:

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

under the bounds

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

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

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

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

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

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.

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

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.
#           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())

```

(continues on next page)

```

# 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')

# 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

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

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

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

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:

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

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:

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

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:

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

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:

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

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 *

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

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

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

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

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

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

MOSEK also supports the dual exponential cone:

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

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:

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

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

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

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:

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

The conic form of (7.5) is:

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

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('ce01') 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('ce01.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

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

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

MOSEK can solve semidefinite optimization problems of the form

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

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.

We demonstrate the setup of semidefinite variables and the matrices \bar{C} , \bar{A} on the following examples:

- [Sec. 7.5.1](#): A problem with one semidefinite variable and linear and conic constraints.
- [Sec. 7.5.2](#): A problem with two semidefinite variables with a linear constraint and bound.
- [Sec. 7.5.3](#): Shows how to efficiently set up many semidefinite variables of the same dimension.

7.5.1 Example SDO1

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

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

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

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

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

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

subject to the two linear constraints

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

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

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

We create a symmetric semidefinite variable \bar{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

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

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())

```

7.5.2 Example SDO2

We now demonstrate how to define more than one semidefinite variable using the following problem with two matrix variables and two types of constraints:

$$\begin{aligned}
 & \text{minimize} && \langle C_1, \overline{X}_1 \rangle + \langle C_2, \overline{X}_2 \rangle \\
 & \text{subject to} && \langle A_1, \overline{X}_1 \rangle + \langle A_2, \overline{X}_2 \rangle = b, \\
 & && (\overline{X}_2)_{01} \leq k, \\
 & && \overline{X}_1, \overline{X}_2 \succeq 0.
 \end{aligned} \tag{7.8}$$

In our example $\dim(\overline{X}_1) = 3$, $\dim(\overline{X}_2) = 4$, $b = 23$, $k = -3$ and

$$C_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 6 \end{bmatrix}, A_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 2 \end{bmatrix},$$

$$C_2 = \begin{bmatrix} 1 & -3 & 0 & 0 \\ -3 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 \end{bmatrix},$$

are constant symmetric matrices.

Note that this problem does not contain any scalar variables, but they could be added in the same fashion as in [Sec. 7.5.1](#).

The code representing the above problem is shown below.

Listing 7.6: Implementation of model (7.8).

```
# Convert input data into Fusion sparse matrices
C1 = Matrix.sparse(3, 3, C1_k, C1_l, C1_v)
C2 = Matrix.sparse(4, 4, C2_k, C2_l, C2_v)
A1 = Matrix.sparse(3, 3, A1_k, A1_l, A1_v)
A2 = Matrix.sparse(4, 4, A2_k, A2_l, A2_v)

# Define the model
with Model('sdo2') as M:
    # Two semidefinite variables
    X1 = M.variable(Domain.inPSDCone(3))
    X2 = M.variable(Domain.inPSDCone(4))

    # Objective
    M.objective(ObjectiveSense.Minimize, Expr.add(Expr.dot(C1,X1), Expr.dot(C2,X2)))

    # Equality constraint
    M.constraint(Expr.add([Expr.dot(A1,X1), Expr.dot(A2,X2)]), Domain.equalsTo(b))

    # Inequality constraint
    M.constraint(X2.index([0,1]), Domain.lessThan(k))

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

    # Retrieve result
    print("X1:\n{0}".format(np.reshape(X1.level(), (3,3))))
    print("X2:\n{0}".format(np.reshape(X2.level(), (4,4))))
```

7.5.3 Example SDO3

Here we demonstrate how to use the facilities provided in *Fusion* to set up a model with many semidefinite variables of the same dimension more efficiently than via looping. We consider a problem with n semidefinite variables of dimension d and k constraints:

$$\begin{aligned} & \text{minimize} && \sum_j \text{tr}(\bar{X}_j) \\ & \text{subject to} && \sum_j \langle A_{ij}, \bar{X}_j \rangle \geq b_i, \quad i = 1, \dots, k, \\ & && \bar{X}_j \succeq 0 \quad j = 1, \dots, n, \end{aligned} \tag{7.9}$$

with symmetric data matrices A_{ij} .

The key construction is:

Listing 7.7: Creating a stack of semidefinite variables.

```
X = M.variable(Domain.inPSDCone(d, n))
```

It creates n symmetric, semidefinite matrix variables of dimension d arranged in a single variable object X of shape (n, d, d) . Individual matrix variables can be accessed as slices from $(i, 0, 0)$ to $(i+1, d, d)$ (reshaped into shape (d, d) if necessary). It is also possible to operate on the full variable X when

constructing expressions that involve entries of all the semidefinite matrices in a natural way. The source code example illustrates both these approaches.

Listing 7.8: Implementation of model (7.9).

```
# Sample input data
n = 100
d = 4
k = 3
b = [9,10,11]
A = list(map(lambda x: x+np.transpose(x),
             [np.random.normal(0, 5, size=(d, d)) for _ in range(k*n)]))

# Create a model with n semidefinite variables of dimension d x d
with Model("sdo3") as M:
    X = M.variable(Domain.inPSDCone(d, n))

    # Pick indexes of diagonal entries for the objective
    alldiag = [[j,s,s] for j in range(n) for s in range(d)]
    M.objective(ObjectiveSense.Minimize, Expr.sum( X.pick(alldiag) ))

    # Each constraint is a sum of inner products
    # Each semidefinite variable is a slice of X
    for i in range(k):
        M.constraint(Expr.add([Expr.dot(A[i*n+j],
                                         X.slice([j,0,0], [j+1,d,d]).reshape([d,d]))
                               for j in range(n)]),
                    Domain.greaterThan(b[i]))

    # Solve
    M.setLogHandler(sys.stdout)          # Add logging
    M.writeTask("sdo3.ptf")              # Save problem in readable format
    M.solve()

    # Get results. Each variable is a slice of X
    print("Contributing variables:")
    for j in range(n):
        Xj = X.slice([j,0,0], [j+1,d,d]).level()
        if any(Xj[s]>1e-6 for s in range(d*d)):
            print("X{0}=\n{1}".format(j, np.reshape(Xj,(d,d))))
```

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 constrained 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

$$\begin{aligned} & \text{maximize} && x_0 + 0.64x_1 \\ & \text{subject to} && 50x_0 + 31x_1 \leq 250, \\ & && 3x_0 - 2x_1 \geq -4, \\ & && x_0, x_1 \geq 0 \quad \text{and integer} \end{aligned} \tag{7.10}$$

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 7.9](#).

Listing 7.9: How to solve problem (7.10).

```
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))
```

(continues on next page)

```

# 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.

$$\begin{aligned}
 &\text{maximize} && 7x_0 + 10x_1 + x_2 + 5x_3 \\
 &\text{subject to} && x_0 + x_1 + x_2 + x_3 \leq 2.5 \\
 & && x_0, x_1, x_2 \in \mathbb{Z} \\
 & && x_0, x_1, x_2, x_3 \geq 0
 \end{aligned} \tag{7.11}$$

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

Listing 7.10: Implementation of problem (7.11) 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 7.11: 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:

$$\begin{aligned}
 &\text{minimize} && x^2 + y^2 \\
 &\text{subject to} && x \geq e^y + 3.8, \\
 &&& x, y \text{ integer.}
 \end{aligned} \tag{7.12}$$

The canonical conic formulation of (7.12) suitable for Fusion API for Python is

$$\begin{aligned}
 &\text{minimize} && t \\
 &\text{subject to} && (t, x, y) \in \mathcal{Q}^3 && (t \geq \sqrt{x^2 + y^2}) \\
 &&& (x - 3.8, 1, y) \in K_{\text{exp}} && (x - 3.8 \geq e^y) \\
 &&& x, y \text{ integer,} \\
 &&& t \in \mathbb{R}.
 \end{aligned} \tag{7.13}$$

Listing 7.12: Implementation of problem (7.13).

```

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

$$\begin{aligned}
 &\text{minimize} && f_0(x) \\
 &\text{subject to} && f_i(x) \leq 1, \quad i = 1, \dots, m, \\
 &&& x_j > 0, \quad j = 1, \dots, n,
 \end{aligned} \tag{7.14}$$

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

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

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

$$\log\left(\sum_k \exp(a_k^T y + b_k)\right) \leq 0 \quad (7.15)$$

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 :

$$\begin{aligned} & \text{maximize} && hwd \\ & \text{subject to} && 2(hw + hd) \leq A_{\text{wall}}, \\ & && wd \leq A_{\text{floor}}, \\ & && \alpha \leq h/w \leq \beta, \\ & && \gamma \leq d/w \leq \delta. \end{aligned} \quad (7.16)$$

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 (7.16) becomes

$$\begin{aligned} & \text{maximize} && x + y + z \\ & \text{subject to} && \log(\exp(x + y + \log(2/A_{\text{wall}})) + \exp(x + z + \log(2/A_{\text{wall}}))) \leq 0, \\ & && y + z \leq \log(A_{\text{floor}}), \\ & && \log(\alpha) \leq x - y \leq \log(\beta), \\ & && \log(\gamma) \leq z - y \leq \log(\delta). \end{aligned} \quad (7.17)$$

Next, we demonstrate how to implement a log-sum-exp constraint (7.15). It can be written as:

$$\begin{aligned} u_k &\geq \exp(a_k^T y + b_k), \quad (\text{equiv. } (u_k, 1, a_k^T y + b_k) \in K_{\text{exp}}), \\ \sum_k u_k &= 1. \end{aligned} \quad (7.18)$$

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

Listing 7.13: Implementation of log-sum-exp as in (7.18).

```
# Models  $\log(\sum(\exp(Ax+b))) \leq 0$ .
# 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 7.14: Source code solving problem (7.17).

```
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 7.15: Sample data for problem (7.16).

```
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 7.16: 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 7.17: 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 7.18: 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 7.19: 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 7.20: 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 7.21: 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 ≥ 1/x^p, x > 0, p > 0
```

Listing 7.22: 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 7.23: 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

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

Listing 7.24: 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)
```

(continues on next page)

(continued from previous page)

```
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

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

Listing 7.25: 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 7.26: Exponential.

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

Entropy

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

Listing 7.27: 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

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

Listing 7.28: 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 7.29: 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, u
↪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 7.30: 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 7.31: 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 7.32: 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 7.33: 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 7.34: 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

$$\begin{aligned} & \text{maximize} && -\sqrt{x^2 + y^2} + \log y - x^{1.5}, \\ & \text{subject to} && x \geq y + 3, \end{aligned} \tag{7.19}$$

or equivalently

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

Listing 7.35: Modeling (7.19).

```
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))
    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 Model Parametrization and Reoptimization

This tutorial demonstrates how to construct a model with a fixed structure and reoptimize it by changing some of the input data. If you instead want to dynamically modify the model structure between optimizations by adding variables, constraints etc., see the other reoptimization tutorial [Sec. 7.10](#).

For this tutorial we solve the following variant of *linear regression with elastic net regularization*:

$$\text{minimize}_x \|Ax - b\|_2 + \lambda_1 \|x\|_1 + \lambda_2 \|x\|_2$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$. The optimization variable is $x \in \mathbb{R}^n$ and λ_1, λ_2 are two nonnegative numbers indicating the tradeoff between the linear regression objective, a *lasso* (ℓ_1 -norm) penalty and a *ridge* (ℓ_2 -norm) regularization. The representation of this problem compatible with **MOSEK** input format is

$$\begin{aligned} & \text{minimize} && t + \lambda_1 \sum_i p_i + \lambda_2 q \\ & \text{subject to} && (t, Ax - b) \in \mathcal{Q}^{m+1}, \\ & && p_i \geq |x_i|, \quad i = 1, \dots, n, \\ & && (q, x) \in \mathcal{Q}^{n+1}. \end{aligned}$$

7.9.1 Creating a model

Before creating a parametrized model we should analyze which parts of the model are fixed once for all, and which parts do we intend to change between optimizations. Here we make the following assumption:

- the matrix A will not change,
- we want to solve the problem for many target vectors b ,
- we want to experiment with different tradeoffs λ_1, λ_2 .

That leads us to construct the model with A provided from the start as fixed input and declare b, λ_1, λ_2 as parameters. The initial model construction is shown below. Parameters are objects of type *Parameter*, created with the method *Model.parameter*. We exploit the fact that parameters can have shapes, just like variables and expressions, and that they can be used everywhere within an expression where a constant of the same shape would be suitable.

Listing 7.36: Constructing a parametrized model.

```
def initializeModel(m, n, A):
    M = Model()
    x = M.variable("x", n)

    # t >= ||Ax-b||_2 where b is a parameter
    b = M.parameter("b", m)
    t = M.variable()
    M.constraint(Expr.vstack(t, Expr.sub(Expr.mul(A, x), b)), Domain.inQCone())

    # p_i >= |x_i|, i=1..n
    p = M.variable(n)
    M.constraint(Expr.hstack(p, x), Domain.inQCone())

    # q >= ||x||_2
    q = M.variable()
    M.constraint(Expr.vstack(q, x), Domain.inQCone())

    # Objective, parametrized with lambda1, lambda2
    # t + lambda1*sum(p) + lambda2*q
    lambda1 = M.parameter("lambda1")
    lambda2 = M.parameter("lambda2")
    obj = Expr.add([t, Expr.mul(lambda1, Expr.sum(p)), Expr.mul(lambda2, q)])
    M.objective(ObjectiveSense.Minimize, obj)
```

(continues on next page)

```
# Return the ready model
return M
```

For the purpose of the example we take

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

and we initialize the parametrized model:

Listing 7.37: Initializing the model

```
# Create a small example
m, n = 4, 2
A = [ [1.0, 2.0],
      [3.0, 4.0],
      [-2.0, -1.0],
      [-4.0, -3.0] ]
M = initializeModel(m, n, A)

# For convenience retrieve some elements of the model
b = M.getParameter("b")
lambda1 = M.getParameter("lambda1")
lambda2 = M.getParameter("lambda2")
x = M.getVariable("x")
```

We made sure to keep references to the interesting elements of the model, in particular the parameter objects we are about to set values of.

7.9.2 Setting parameters

For the first solve we use

$$b = [0.1, 1.2, -1.1, 3.0]^T, \lambda_1 = 0.1, \lambda_2 = 0.01.$$

Parameters are set with method `Parameter.setValue`. We set the parameters and solve the model as follows:

Listing 7.38: Setting parameters and solving the model.

```
# First solve
b.setValue([0.1, 1.2, -1.1, 3.0])
lambda1.setValue(0.1)
lambda2.setValue(0.01)

M.solve()
print("Objective {0}, solution x = {1}".format(M.primalObjValue(), x.level()))
```

7.9.3 Changing parameters

Let us say we now want to increase the weight of the lasso penalty in order to favor sparser solutions. We can simply change that parameter, leave the other ones unchanged, and resolve:

Listing 7.39: Changing a parameter and resolving

```
# Increase lambda1
lambda1.setValue(0.5)

M.solve()
print("Objective {0}, solution x = {1}".format(M.primalObjValue(), x.level()))
```

Next, we might want to solve a few instances of the problem for another value of b . Again, we reset the relevant parameters and solve:

Listing 7.40: Changing parameters and resolving

```
# Now change the data completely
b.setValue([1.0, 1.0, 1.0, 1.0])
lambda1.setValue(0.0)
lambda2.setValue(0.0)

M.solve()
print("Objective {0}, solution x = {1}".format(M.primalObjValue(), x.level()))

# And increase lambda2
lambda2.setValue(1.4145)

M.solve()
print("Objective {0}, solution x = {1}".format(M.primalObjValue(), x.level()))
```

7.9.4 Additional remarks

- Domains cannot be parametrized, therefore to parametrize a bound, such as $x \geq p$, it is necessary to write it as $x - p \geq 0$.
- Coefficients appearing at semidefinite terms cannot be parametrized. If it is necessary to have a parametrized expression such as $p\bar{X}_{i,j}$, introduce an auxiliary scalar variable $x = \bar{X}_{i,j}$ and use px in the model.
- Parametrized models can be found in the following examples: `alan.py`, `portfolio_2_frontier.py`, `portfolio_5_card.py`, `total_variation.py`.

7.10 Problem Modification and Reoptimization

This tutorial demonstrates how to modify a model by adding new elements and changing existing ones. If instead you want to create one model of fixed structure and reoptimize it for changing input data, see [Sec. 7.9](#).

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 [[Chvatal83](#)].

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

7.10.1 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:

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

and

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

Code in [Listing 7.41](#) loads and solves this problem.

Listing 7.41: Setting up and solving problem (7.20)

```
# 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()
```

7.10.2 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:

$$\begin{array}{llllll} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \quad (7.21)$$

and

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

After this operation we can reoptimize the problem.

7.10.3 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 7.42

Listing 7.42: 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 existing constraint
con.update(Expr.mul(x3, [4, 0, 1]), x3)
```

(continues on next page)

(continued from previous page)

```
# Change the objective to include x3
M.objective(ObjectiveSense.Maximize, Expr.dot(c+[1.0], xNew))
```

After this operation the new problem is:

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

and

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

7.10.4 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 7.43: 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.

7.10.5 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:

$$\begin{array}{llllllll}
\text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 \\
\text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 80000, \\
& 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 40000, \\
& 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 50000, \\
& x_0 & + & 2x_1 & + & x_2 & + & x_3 & \leq & 22000.
\end{array} \tag{7.23}$$

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:

$$\begin{aligned}
 &\text{maximize} && 1.5x_0 &+& 2.5x_1 &+& 3.0x_2 &+& 1.0x_3 \\
 &\text{subject to} && 3x_0 &+& 4x_1 &+& 3x_2 &+& 4x_3 &+& 20000 &\leq & 100000, \\
 &&& 3x_0 &+& 2x_1 &+& 3x_2 && &+& 10000 &\leq & 50000, \\
 &&& 2x_0 &+& 3x_1 &+& 2x_2 &+& 1x_3 &+& 10000 &\leq & 60000, \\
 &&& x_0 &+& 2x_1 &+& x_2 &+& x_3 &+& 8000 &\leq & 30000.
 \end{aligned} \tag{7.24}$$

The next listing shows how to do it.

Listing 7.44: 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.

7.10.6 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	<code>SolutionType.Basic</code>	<code>SolutionType.Interior</code>	
Conic (nonlinear) problem		<code>SolutionType.Interior</code>	
Problem with integer variables			<code>SolutionType.Integer</code>

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	<i>ProblemStatus.PrimalAndDualFeasible</i>	<i>SolutionStatus.Optimal</i>	<i>SolutionStatus.Optimal</i>
Primal infeasible	<i>ProblemStatus.PrimalInfeasible</i>	<i>SolutionStatus.Unknown</i>	<i>SolutionStatus.Certificate</i>
Dual infeasible (unbounded)	<i>ProblemStatus.DualInfeasible</i>	<i>SolutionStatus.Certificate</i>	<i>SolutionStatus.Unknown</i>
Uncertain (stall, numerical issues, etc.)	<i>ProblemStatus.Unknown</i>	<i>SolutionStatus.Unknown</i>	<i>SolutionStatus.Unknown</i>

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

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

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 diagnose 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:
```

(continues on next page)

(continued from previous page)

```
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 diagnose why.
        print("Requested solution was not available.")
        prosta = M.getProblemStatus()

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

(continues on next page)

```

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))

```

8.3 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.is_alive():
            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 userProgressCallback(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:
```

(continues on next page)

```

itrn = intinf[iinfitem.intpnt_iter]
pobj = douinf[dinfitem.intpnt_primal_obj]
dobj = douinf[dinfitem.intpnt_dual_obj]
stime = douinf[dinfitem.intpnt_time]
opttime = douinf[dinfitem.optimizer_time]

print("Iterations: %-3d" % itrn)
print("  Elapsed time: %6.2f(%.2f)" % (opttime, stime))
print("  Primal obj.: %-18.6e  Dual obj.: %-18.6e" % (pobj, dobj))
elif caller == callbackcode.end_intpnt:
    print("Interior-point optimizer finished.")
elif caller == callbackcode.begin_primal_simplex:
    print("Primal simplex optimizer started.")
elif caller == callbackcode.update_primal_simplex:
    itrn = intinf[iinfitem.sim_primal_iter]
    pobj = douinf[dinfitem.sim_obj]
    stime = douinf[dinfitem.sim_time]
    opttime = douinf[dinfitem.optimizer_time]

    print("Iterations: %-3d" % itrn)
    print("  Elapsed time: %6.2f(%.2f)" % (opttime, stime))
    print("  Obj.: %-18.6e" % pobj)
elif caller == callbackcode.end_primal_simplex:
    print("Primal simplex optimizer finished.")
elif caller == callbackcode.begin_dual_simplex:
    print("Dual simplex optimizer started.")
elif caller == callbackcode.update_dual_simplex:
    itrn = intinf[iinfitem.sim_dual_iter]
    pobj = douinf[dinfitem.sim_obj]
    stime = douinf[dinfitem.sim_time]
    opttime = douinf[dinfitem.optimizer_time]
    print("Iterations: %-3d" % itrn)
    print("  Elapsed time: %6.2f(%.2f)" % (opttime, stime))
    print("  Obj.: %-18.6e" % pobj)
elif caller == callbackcode.end_dual_simplex:
    print("Dual simplex optimizer finished.")
elif caller == callbackcode.begin_bi:
    print("Basis identification started.")
elif caller == callbackcode.end_bi:
    print("Basis identification finished.")
else:
    pass

if opttime >= maxtime:
    # mosek is spending too much time. Terminate it.
    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.

8.9 MOSEK OptServer

MOSEK provides an easy way to offload optimization problem to a remote server. This section demonstrates related functionalities from the client side, i.e. sending optimization tasks to the remote server and retrieving solutions.

Setting up and configuring the remote server is described in a separate manual for the OptServer.

8.9.1 Synchronous Remote Optimization

In synchronous mode the client sends an optimization problem to the server and blocks, waiting for the optimization to end. Once the result has been received, the program can continue. This is the simplest mode all it takes is to provide the address of the server before starting optimization. The rest of the code remains untouched.

Note that it is impossible to recover the job in case of a broken connection.

Source code example

Listing 8.8: Using the OptServer in synchronous mode.

```
from mosek.fusion import *
import sys

if len(sys.argv) < 1:
    print("Missing argument, syntax is:")
    print("  python opt_server_sync.py serveraddr")
    sys.exit(1)

addr = sys.argv[1]

with Model('testOptServer') as M:
    # Setup a simple test problem
    x = M.variable('x', 3, Domain.greaterThan(0.0))
    M.constraint("lc", Expr.dot([1.0, 1.0, 2.0], x), Domain.equalsTo(1.0))
    M.objective("obj", ObjectiveSense.Minimize, Expr.sum(x))

    # Attach log handler
    M.setLogHandler(sys.stdout)
```

(continues on next page)

(continued from previous page)

```
# Set OptServer URL
M.optserverHost(addr)

# Solve the problem on the OptServer
M.solve()

# Get the solution
print('x1,x2,x3 = %s' % str(x.level()))
```

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 \mathbf{x} , 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 $\text{dist}(x, \mathcal{K})$.
- 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:

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

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

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

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](#).

The problem represented in [Fig. 9.1](#) is infeasible, since the total demand

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

exceeds the total supply

$$2200 = 200 + 1000 + 1000$$

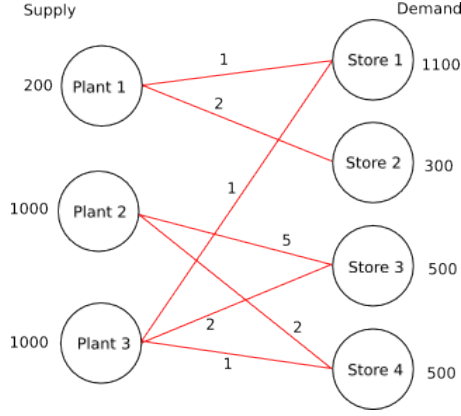


Fig. 9.1: Supply, demand and cost of transportation.

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:

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

Solving problem (9.1) using **MOSEK** will result in an infeasibility status. The infeasibility certificate is contained in the dual variables and 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

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

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					
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					

(continues on next page)

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 y_1, y_3, y_4, y_5 and two constraints contribute to infeasibility with non-zero values in the **Activity** column. Therefore

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

is the dual infeasibility certificate as in [Sec. 12.1.2](#). This just means, that along the ray

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

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 y_3 . This will directly invalidate the certificate of dual infeasibility.
- Increase the objective coefficient of y_3 . 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

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

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

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

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
paramdiff	Show parameters with non-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
resobj [fac]	Rescale objective by a factor
anasol	Analyze solutions
removeitg	Remove integrality constraints
removecones	Remove all cones and leave just the linear part
infsub	Replace current problem with its infeasible subproblem
writesol basename	Write solution(s) to file(s) with given basename
del_sol	Remove all solutions from the task
optserver [url]	Use an OptServer to optimize
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 maximum number of structural nonzeros in any single expression object 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.

Names introduced in *Fusion* are transformed into names in the underlying low-level optimization task, which in turn can be saved to a file. In particular:

- a scalar variable with name `var` becomes a variable with name `var[]`,
- a one- or more-dimensional variable with name `var` becomes a sequence of scalar variables with names `var[0]`, `var[1]`, etc. or `var[0][0]`, `var[0][1]`, etc., depending on the shape,
- the same applies to constraints,
- for a conic constraint with name `con` a sequence of slack variables with names `con[0].coneslack`, etc. or `con[0][0].coneslack`, etc., depending on the shape of the constraint, is added.
- a new variable with name `1.0` may be added.

These are the guidelines. No guarantees are made for the exact form of this transformation.

Note that file formats impose various restrictions on names, so not all resulting names can be written verbatim to each type of file, and when writing to a file further transformations and character substitutions can be applied, resulting in poor readability. This is particularly true for LP files, so saving *Fusion* problems in LP format is discouraged. The OPF format is recommended instead. See [Sec. 15](#).

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 an 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. 6.8](#) for more examples.

Parametrize relevant parts of the model

If you intend to reoptimize the same model with changing input data, use a parametrized model and modify it between optimizations by resetting parameter values, see [Sec. 6.6](#). This way the model is constructed only once, and only a few coefficients need to be recomputed each time.

Keep a healthy balance and parametrize only the part of the model you in fact intend to change. For example, using parameters in place of all constants appearing in the model would be an overkill with an adverse effect on efficiency since all coefficients in the problem would still have to be recomputed each time.

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. 6.7](#). 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 9.3.21, the following libraries must be included:

64-bit x86 Linux	64-bit Windows	32-bit Windows	64-bit Mac OS	Linux ARM64
libmosek64.so.9.3	mosek64_9_3.dll	mosek9_3.dll	libmosek64.9.3.dylib	libmosek64.so.9.3
libcilkrts.so.5	cilkrts20.dll	cilkrts20.dll	libcilkrts.5.dylib	libmosekxx9_3.so
libmosekxx9_3.so	mosekxx9_3.dll	mosekxx9_3.dll	libmosekxx9_3.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, Model parametrization
- *Primal SVM*
 - **Keywords:** machine learning, Support-Vector Machine, hyperplane separation, classifier
 - **Type:** Conic Quadratic
- *2D Total Variation*
 - **Keywords:** denoising, total variation
 - **Type:** Conic Quadratic, Model parametrization
- *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

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

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

$$\begin{aligned} x^T \Sigma x &= x^T GG^T x \\ &= \|G^T x\|^2. \end{aligned}$$

Hence, we may write the risk constraint as

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

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

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

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
```

(continues on next page)

(continued from previous page)

```

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

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

# Solves the model.
M.solve()

return 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

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

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

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

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

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

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:

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

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:

```

(continues on next page)

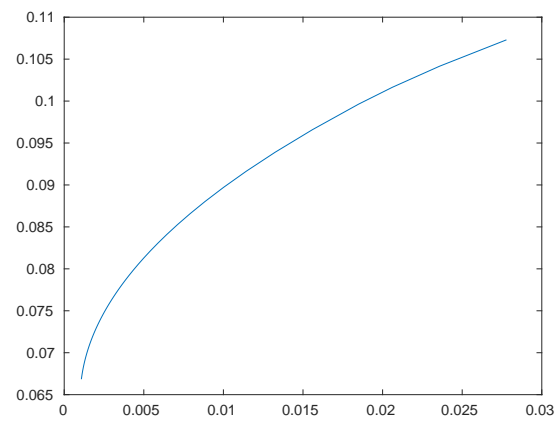


Fig. 11.1: The efficient frontier for the sample data.

```

frontier = []

# 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

# Total budget constraint
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())

# Define objective as a weighted combination of return and variance
alpha = M.parameter()
M.objective('obj', ObjectiveSense.Maximize, Expr.sub(Expr.dot(mu,x), Expr.
→mul(alpha,s)))

# Solve multiple instances by varying the parameter alpha
for a in alphas:
    alpha.setValue(a);

    M.solve()

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

return frontier

```

Note that we defined α as a model parameter and used it to parametrize the objective. This way we were able to reuse the same model for all solves along the efficient frontier, simply changing the value of α between the solves.

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

$$\begin{aligned} & \text{maximize} && \mu^T x \\ & \text{subject to} && e^T x + \sum_{j=1}^n T_j(\Delta x_j) = w + e^T x^0, \\ & && x^T \Sigma x \leq \gamma^2, \\ & && x \geq 0. \end{aligned} \tag{11.6}$$

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

$$\begin{aligned} z_j &= |x_j - x_j^0|, \\ (t_j, 1, z_j) &\in \mathcal{P}_3^{2/3, 1/3}. \end{aligned}$$

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

$$\begin{aligned} & \text{maximize} && \mu^T x \\ & \text{subject to} && e^T x + m^T t = w + e^T x^0, \\ & && (\gamma, G^T x) \in \mathcal{Q}^{n+1}, \\ & && (t_j, 1, x_j - x_j^0) \in \mathcal{P}_3^{2/3, 1/3}, \quad j = 1, \dots, n, \\ & && x \geq 0. \end{aligned} \tag{11.10}$$

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))
```

(continues on next page)

Example code

The following example code demonstrates how to compute an optimal portfolio with cardinality bounds. Note that we define the maximum cardinality as a parameter in the model and use it to parametrize the cardinality constraint. This way we can use one model to solve many problems with the same structure and data except for the cardinality bound by simply changing this parameter between the solves.

Listing 11.5: Code solving problem (11.12).

```
def MarkowitzWithCardinality(n,mu,GT,x0,w,gamma,kValues):
    # 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))

        # 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
        cardMax = M.parameter()
        M.constraint('cardinality', Expr.sub(Expr.sum(y), cardMax), Domain.
↪lessThan(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)

        # Solve multiple instances by varying the parameter k
        results = []
        for k in kValues:
            cardMax.setValue(k)
            M.solve()
            results.append(x.level())

    return results
```


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())
```

(continues on next page)

```

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
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.

For $\sigma = 1$ the two groups are not linearly separable and the we obtain the optimal hyperplane as in Fig. 11.3.

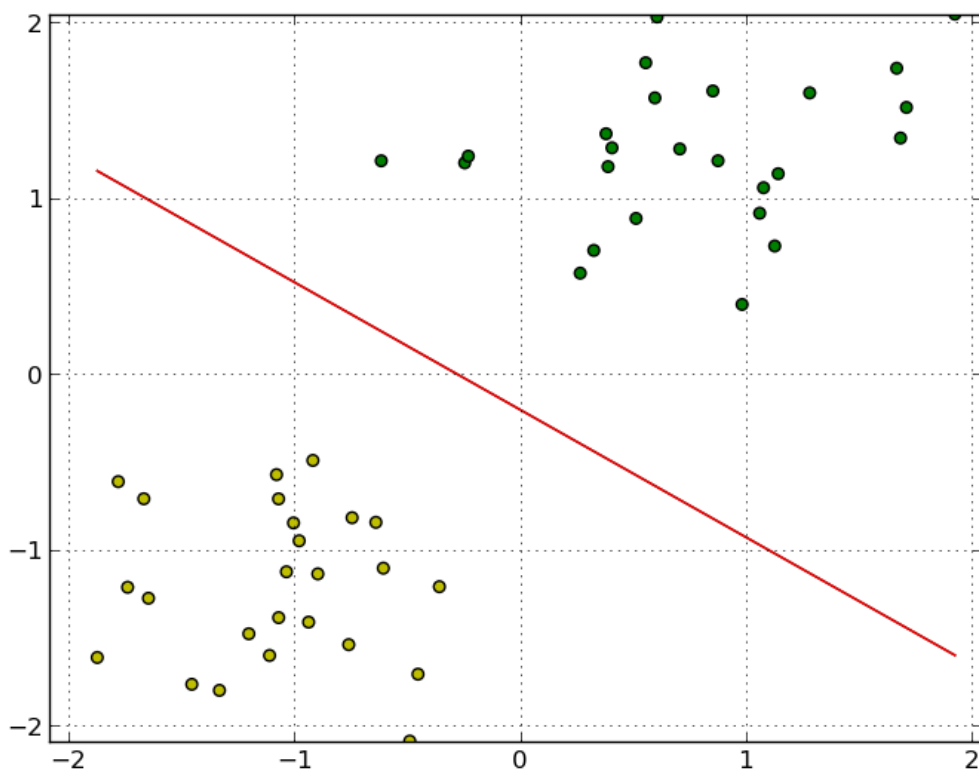


Fig. 11.2: Separating hyperplane for two clusters of points.

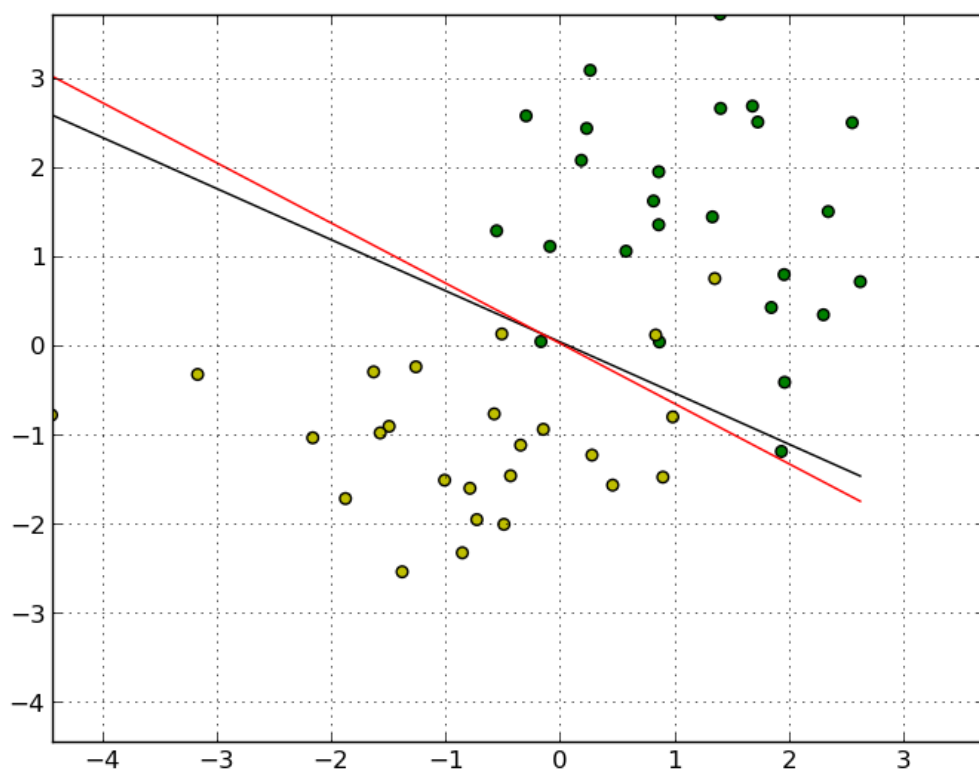


Fig. 11.3: Soft separating hyperplane for two groups of points.


```

if __name__ == '__main__':
    np.random.seed(0)

    n, m = 100, 200

    # Create a parametrized model with given shape
    M = total_var(n, m)
    sigma = M.getParameter("sigma")
    f      = M.getParameter("f")
    ucore = M.getVariable("u").slice([0,0], [n,m])

    # Example: Linear signal with Gaussian noise
    signal = np.reshape([[1.0*(i+j)/(n+m) for i in range(m)] for j in range(n)], (n,
↪m))
    noise  = np.random.normal(0., 0.08, (n,m))
    fVal   = signal + noise

    # Uncomment to get graphics:
    # show(n, m, signal)
    # show(n, m, fVal)

    # Set value for f
    f.setValue(fVal)

    for sigmaVal in [0.0004, 0.0005, 0.0006]:
        # Set new value for sigma and solve
        sigma.setValue(sigmaVal*n*m)

        M.solve()

        sol = np.reshape(ucore.level(), (n,m))
        # Now use the solution
        # ...

        # Uncomment to get graphics:
        # show(n, m, np.reshape(ucore.level(), (n,m)))

        print("rel_sigma = {sigmaVal}  total_var = {var}".format(sigmaVal=sigmaVal,
                                                                    var=M.
↪primalObjValue() ))

    M.dispose()

```

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.


```

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
        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()])

```


- **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*) – An auxiliary stack.

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

LinearConstraint.toString

```
toString() -> str
```

Create a human readable string representation of the constraint.

Return (str)

14.2.14 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  
withShape(int dim0) -> LinearDomain  
withShape(int dim0, int dim1) -> LinearDomain  
withShape(int dim0, int dim1, int dim2) -> LinearDomain
```

Set the shape of the domain.

Parameters

- shp (int[]) – The shape of the domain

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*)

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*

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 degeneracy strategy.
"free"	The simplex optimizer chooses the degeneracy strategy.
"aggressive"	The simplex optimizer should use an aggressive degeneracy strategy.
"moderate"	The simplex optimizer should use a moderate degeneracy strategy.
"minimum"	The simplex optimizer should use a minimum degeneracy 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.

"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.

"beginQcqrReformulate"
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.

"endQcqpReformulate"
End QCQP 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.

"imDualSensitivity"
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.

"imPrimalSensitivity"
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.

"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.

"rdPrototype"
 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.

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).

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

- *DeletionError*: This item cannot be removed.
- *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 patterns was invalid or specified an index that was out of bounds.
- *UnexpectedError*: An unexpected error has happened. No specific exception could have been risen.
- *UnimplementedError*: Called a stub. Functionality has not yet been implemented.
- *UpdateError*: Invalid slice definition, negative slice or slice index out of bounds.
- *ValueConversionError*: Error casting or converting a value.

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. It must not overlap with the other input arrays.

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.

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

$$\begin{array}{ll} \text{minimize/maximize} & c^T x + \frac{1}{2} q^o(x) \\ \text{subject to} & \begin{array}{ll} l^c \leq & Ax + \frac{1}{2} q(x) \leq u^c, \\ l^x \leq & x \leq u^x, \\ & x_{\mathcal{J}} \text{ integer,} \end{array} \end{array}$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear term in the objective.
- $q^o : \mathbb{R}^n \rightarrow \mathbb{R}$ 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

$$Q^i = (Q^i)^T.$$

- $\mathcal{J} \subseteq \{1, 2, \dots, n\}$ 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.

$$c^T x + \frac{1}{2} x^T Q x.$$

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

$$\text{minimize } 4x_1 + x_2 - 0.1 \cdot x_3 + \frac{1}{2}(x_1^2 + 2.1 \cdot x_1 \cdot x_2)$$

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 \leq) may be any of $<$, \leq , $=$, $>$, \geq ($<$ and \leq 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.

$$-5 \leq x_1 + x_2 \leq 5 \quad (15.1)$$

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

$$x_1 + x_2 - sl_1 = 0, \quad -5 \leq sl_1 \leq 5.$$

Bounds

Bounds on the variables can be specified in the bound section beginning with one of the keywords

```
bound
bounds
```

The bounds section is optional but should, if present, follow the **subject to** section. All variables listed in the bounds section must occur in either the objective or a constraint.

The default lower and upper bounds are 0 and $+\infty$. A variable may be declared free with the keyword **free**, which means that the lower bound is $-\infty$ and the upper bound is $+\infty$. Furthermore it may be assigned a finite lower and upper bound. The bound definitions for a given variable may be written in one or two lines, and bounds can be any number or $\pm\infty$ (written as **+inf/-inf/+infinity/-infinity**) as in the example

```
bounds
x1 free
x2 <= 5
0.1 <= x2
x3 = 42
2 <= x4 < +inf
```

Variable Types

The final two sections are optional and must begin with one of the keywords

```
bin
binaries
binary
```

and

```
gen
general
```

Under **general** all integer variables are listed, and under **binary** all binary (integer variables with bounds 0 and 1) are listed:

```
general
x1 x2
binary
x3 x4
```

Again, all variables listed in the binary or general sections must occur in either the objective or a constraint.

Terminating Section

Finally, an LP formatted file must be terminated with the keyword

```
end
```

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
```


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.


```

[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

$$\begin{aligned}
 &\text{minimize} && x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\
 &\text{subject to} && 1 \leq x_1 + x_2 + x_3, \\
 &&& x \geq 0.
 \end{aligned}$$

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.

  - 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:

$$\begin{array}{llll} \text{minimize} & x_3 + x_4 + x_5 \\ \text{subject to} & x_0 + x_1 + 2x_2 = 1, \\ & x_0, x_1, x_2 \geq 0, \\ & x_3 \geq \sqrt{x_0^2 + x_1^2}, \\ & 2x_4x_5 \geq x_2^2. \end{array}$$

Please note that the type of the cones is defined by the parameter to `[cone ...]`; the content of the `cone`-section is the names of variables that belong to the cone. The resulting OPF file is in [Listing 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
  [cone rquad 'k2'] x5, x6, x3 [/cone]
[/bounds]
```

Mixed Integer Example milo1.opf

Consider the mixed integer problem:

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

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]
```



```

F 3

# Five scalar constraints with affine expressions in two conic domains:
#   | Two are fixed to zero.
#   | Three are in conic quadratic domain.
CON
5 2
L= 2
Q 3

# Five coordinates in  $F^{\{obj\}}_j$  coefficients:
#   |  $F^{\{obj\}}[0][0,0] = 2.0$ 
#   |  $F^{\{obj\}}[0][1,0] = 1.0$ 
#   | and more...
OBJFCOORD
5
0 0 0 2.0
0 1 0 1.0
0 1 1 2.0
0 2 1 1.0
0 2 2 2.0

# One coordinate in  $a^{\{obj\}}_j$  coefficients:
#   |  $a^{\{obj\}}[1] = 1.0$ 
OBJACOORD
1
1 1.0

# Nine coordinates in  $F_{ij}$  coefficients:
#   |  $F[0,0][0,0] = 1.0$ 
#   |  $F[0,0][1,1] = 1.0$ 
#   | and more...
FCOORD
9
0 0 0 0 1.0
0 0 1 1 1.0
0 0 2 2 1.0
1 0 0 0 1.0
1 0 1 0 1.0
1 0 2 0 1.0
1 0 1 1 1.0
1 0 2 1 1.0
1 0 2 2 1.0

# Six coordinates in  $a_{ij}$  coefficients:
#   |  $a[0,1] = 1.0$ 
#   |  $a[1,0] = 1.0$ 
#   | and more...
ACOORD
6
0 1 1.0
1 0 1.0
1 2 1.0
2 1 1.0
3 0 1.0
4 2 1.0

```

(continues on next page)


```

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:
#   |  $a[0,0] = -1.0$ 
#   |  $a[0,1] = -1.0$ 
ACCOORD
2
0 0 -1.0
0 1 -1.0

# Four coordinates in  $H_{ij}$  coefficients:
#   |  $H[0,0][1,0] = 1.0$ 
#   |  $H[0,0][1,1] = 3.0$ 
#   | and more...
HCOORD
4
0 0 1 0 1.0
0 0 1 1 3.0
0 1 0 0 3.0
0 1 1 0 1.0

# Two coordinates in  $D_i$  coefficients:
#   |  $D[0][0,0] = -1.0$ 
#   |  $D[0][1,1] = -1.0$ 
DCCOORD
2
0 0 0 -1.0
0 1 1 -1.0

```



```

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 than 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

```

(continues on next page)


```

        1e+30
    ],
    "type": [
        "cont",
        "cont",
        "cont",
        "cont"
    ]
},
"con": {
    "name": [
        "c1",
        "c2",
        "c3"
    ],
    "bk": [
        "fx",
        "lo",
        "up"
    ],
    "bl": [
        3e+1,
        1.5e+1,
        -1e+30
    ],
    "bu": [
        3e+1,
        1e+30,
        2.5e+1
    ]
},
"objective": {
    "sense": "max",
    "name": "obj",
    "c": {
        "subj": [
            0,
            1,
            2,
            3
        ],
        "val": [
            3e+0,
            1e+0,
            5e+0,
            1e+0
        ]
    },
    "cfix": 0.0
},
"A": {
    "subi": [
        0,
        0,
        0,
        1,

```

```

        1,
        1,
        1,
        2,
        2
    ],
    "subj": [
        0,
        1,
        2,
        0,
        1,
        2,
        3,
        1,
        3
    ],
    "val": [
        3e+0,
        1e+0,
        2e+0,
        2e+0,
        1e+0,
        3e+0,
        1e+0,
        2e+0,
        3e+0
    ]
}
},
"Task/parameters": {
    "iparam": {
        "ANA_SOL_BASIS": "ON",
        "ANA_SOL_PRINT_VIOLATED": "OFF",
        "AUTO_SORT_A_BEFORE_OPT": "OFF",
        "AUTO_UPDATE_SOL_INFO": "OFF",
        "BASIS_SOLVE_USE_PLUS_ONE": "OFF",
        "BI_CLEAN_OPTIMIZER": "OPTIMIZER_FREE",
        "BI_IGNORE_MAX_ITER": "OFF",
        "BI_IGNORE_NUM_ERROR": "OFF",
        "BI_MAX_ITERATIONS": 1000000,
        "CACHE_LICENSE": "ON",
        "CHECK_CONVEXITY": "CHECK_CONVEXITY_FULL",
        "COMPRESS_STATFILE": "ON",
        "CONCURRENT_NUM_OPTIMIZERS": 2,
        "CONCURRENT_PRIORITY_DUAL_SIMPLEX": 2,
        "CONCURRENT_PRIORITY_FREE_SIMPLEX": 3,
        "CONCURRENT_PRIORITY_INTPNT": 4,
        "CONCURRENT_PRIORITY_PRIMAL_SIMPLEX": 1,
        "FEASREPAIR_OPTIMIZE": "FEASREPAIR_OPTIMIZE_NONE",
        "INFEAS_GENERIC_NAMES": "OFF",
        "INFEAS_PREFER_PRIMAL": "ON",
        "INFEAS_REPORT_AUTO": "OFF",
        "INFEAS_REPORT_LEVEL": 1,
        "INTPNT_BASIS": "BI_ALWAYS",
        "INTPNT_DIFF_STEP": "ON",

```


Table 16.1 – continued from previous page

File	Description
<code>portfolio_5_card.py</code>	Portfolio optimization - cardinality constraints
<code>pow1.py</code>	A simple power cone problem
<code>primal_svm.py</code>	Implements a simple soft-margin Support Vector Machine (CQO)
<code>qcqp_sdo_relaxation.py</code>	Demonstrate how to use SDP to solve convex relaxation of a mixed-integer QCQO problem
<code>reoptimization.py</code>	Demonstrate how to modify and re-optimize a linear problem
<code>response.py</code>	Demonstrates proper response handling
<code>sdo1.py</code>	A simple semidefinite problem with one matrix variable and a quadratic cone
<code>sdo2.py</code>	A simple semidefinite problem with two matrix variables
<code>sdo3.py</code>	A simple semidefinite problem with many matrix variables of the same dimension
<code>sospoly.py</code>	Models the cone of nonnegative polynomials and nonnegative trigonometric polynomials using Nesterov's framework
<code>sudoku.py</code>	A SUDOKU solver (MIP)
<code>total_variation.py</code>	Demonstrates how to solve a total variation problem (CQO)
<code>tsp.py</code>	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.

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

