



MOSEK Fusion API for C++
Release 9.3.21

MOSEK ApS

08 August 2022

Contents

1	Introduction	1
1.1	Why the Fusion API for C++?	2
2	Contact Information	3
3	License Agreement	4
4	Installation	8
4.1	Testing the installation and compiling examples	10
4.2	Creating a Visual Studio Project	10
4.3	Troubleshooting	10
5	Design Overview	12
6	Conic Modeling	14
6.1	The model	14
6.2	Variables	15
6.3	Linear algebra	15
6.4	Constraints and objective	16
6.5	Matrices	17
6.6	Parameters	18
6.7	Stacking and views	18
6.8	Vectorization	19
6.9	Reoptimization	20
7	Optimization Tutorials	22
7.1	Linear Optimization	22
7.2	Conic Quadratic Optimization	24
7.3	Power Cone Optimization	27
7.4	Conic Exponential Optimization	30
7.5	Semidefinite Optimization	32
7.6	Integer Optimization	38
7.7	Geometric Programming	41
7.8	Library of basic functions	43
7.9	Model Parametrization and Reoptimization	48
7.10	Problem Modification and Reoptimization	51
8	Solver Interaction Tutorials	55
8.1	Accessing the solution	55
8.2	Errors and exceptions	59
8.3	Input/Output	61
8.4	Setting solver parameters	62
8.5	Retrieving information items	63
8.6	Stopping the solver	64
8.7	Progress and data callback	65
8.8	Optimizer API Task	68
8.9	MOSEK OptServer	68

9	Debugging Tutorials	70
9.1	Understanding optimizer log	70
9.2	Addressing numerical issues	75
9.3	Debugging infeasibility	77
9.4	Python Console	81
10	Technical guidelines	84
10.1	Limitations	84
10.2	Memory management and garbage collection	84
10.3	Names	85
10.4	Multithreading	85
10.5	Efficiency	86
10.6	The license system	87
10.7	Deployment	88
11	Case Studies	89
11.1	Portfolio Optimization	90
11.2	Primal Support-Vector Machine (SVM)	102
11.3	2D Total Variation	105
11.4	Multiprocessor Scheduling	111
11.5	Logistic regression	115
11.6	Inner and outer Löwner-John Ellipsoids	117
11.7	SUDOKU	121
11.8	Travelling Salesman Problem (TSP)	126
11.9	Nearest Correlation Matrix Problem	131
11.10	Semidefinite Relaxation of MIQCQO Problems	135
12	Problem Formulation and Solutions	139
12.1	Linear Optimization	139
12.2	Conic Optimization	142
12.3	Semidefinite Optimization	146
13	Optimizers	148
13.1	Presolve	148
13.2	Linear Optimization	150
13.3	Conic Optimization - Interior-point optimizer	157
13.4	The Optimizer for Mixed-integer Problems	161
14	<i>Fusion</i> API Reference	165
14.1	<i>Fusion</i> API conventions	165
14.2	Class list	171
14.3	Parameters grouped by topic	253
14.4	Parameters (alphabetical list sorted by type)	260
14.5	Enumerations	286
14.6	Constants	287
14.7	Exceptions	310
14.8	Class LinAlg	314
15	Supported File Formats	319
15.1	The LP File Format	320
15.2	The MPS File Format	325
15.3	The OPF Format	337
15.4	The CBF Format	347
15.5	The PTF Format	361
15.6	The Task Format	366
15.7	The JSON Format	366
15.8	The Solution File Format	374
16	List of examples	377

17 Interface changes	379
17.1 Backwards compatibility	379
17.2 Parameters	380
17.3 Constants	381
Bibliography	382
Symbol Index	383
Index	386

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 C++?

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 C++ 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 C++.

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 C++ is compatible with the following compiler tool chains:

Platform	Supported compiler	Framework
Linux 64 bit x86	gcc (≥ 4.8)	glibc (≥ 2.2)
Mac OS 64 bit	Xcode (≥ 5)	MAC OS SDK (≥ 10.7)
Windows 64 bit	Visual Studio (≥ 2015)	Windows 8.1 SDK
Linux ARM64	gcc (≥ 9.2)	glibc (≥ 2.20)

In many cases older versions can also be used. In particular Fusion API for C++ requires a C++11 compliant compiler.

Locating files in the MOSEK Optimization Suite

The relevant files of the Fusion API for C++ are organized as reported in [Table 4.1](#).

Table 4.1: Relevant files for the Fusion API for C++.

Relative Path	Description	Label
<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/h	Header files	<HEADERDIR>
<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/src/fusion_cxx	Source files	<SRCDIR>
<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/bin	Shared libraries	<LIBDIR>
<MSKHOME>/mosek/9.3/tools/examples/fusion/cxx	Examples	<EXDIR>
<MSKHOME>/mosek/9.3/tools/examples/fusion/data	Additional data	<MISCDIR>

where

- <MSKHOME> is the folder in which the **MOSEK** Optimization Suite has been installed,
- <PLATFORM> is the actual platform among those supported by **MOSEK**, i.e. win64x86, linux64x86, osx64x86 or linuxaarch64.

Manual compilation (all platforms)

This step is compulsory on Linux and Mac OS. The implementation of *Fusion* is distributed as C++ source code in `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/src/fusion_cxx`. It can be compiled as follows:

1. Go to `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/src/fusion_cxx`
2. Run `make install` (Linux, Mac OS) or `nmake install` (Windows).
3. If no error occurs, then the *Fusion* C++ API has been successfully compiled and the corresponding libraries have been copied to `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/bin`.

Using a pre-compiled library (only Windows)

On Windows 64bit the users can skip the compilation step and use a pre-compiled library `fusion64_9_3.lib` available in `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/bin`.

Setting up paths and linking

To compile and link a C++ application using *Fusion*, the user must set paths to header files, compiled *Fusion* libraries, and run-time dependencies must be resolved. Details vary depending on the operating system and compiler. See the `Makefile` included in the distribution under `<MSKHOME>/mosek/9.3/tools/examples/fusion/cxx` for a working example. Typically:

- Linux 64bit x86:

```
g++ -std=c++11 file.cc -o file -I<HEADERDIR> -L<LIBDIR> -Wl,-rpath-link,<LIBDIR> ↵  
↪ -Wl,-rpath=<LIBDIR> -lmosek64 -lfusion64
```

The shared libraries `libmosek64.so.9.3`, `libfusion64.so.9.3` must be available at runtime.

- Mac OS:

```
clang++ -std=c++11 -stdlib=libc++ file.cc -o file -I<HEADERDIR> -L<LIBDIR> -Wl,  
↪ -headerpad,128 -lmosek64 -lfusion64  
install_name_tool -change libmosek64.9.3.dylib <LIBDIR>/libmosek64.9.3.dylib file  
install_name_tool -change libfusion64.9.3.dylib <LIBDIR>/libfusion64.9.3.dylib ↵  
↪ file
```

The shared libraries `libmosek64.9.3.dylib`, `libfusion64.9.3.dylib` must be available at runtime.

- Windows:

```
cl /I<HEADERDIR> file.cc /link /LIBPATH:<LIBDIR> fusion64_9_3.lib mosek64_9_3.lib
```

The shared library `mosek64_9_3.dll` must be available at runtime.

- Linux ARM64:

```
g++ -std=c++11 file.cc -o file -I<HEADERDIR> -L<LIBDIR> -Wl,-rpath-link,<LIBDIR> ↵  
↪ -Wl,-rpath=<LIBDIR> -lpthread -lmosek64 -lfusion64
```

The shared libraries `libmosek64.so.9.3`, `libfusion64.so.9.3` must be available at runtime.

Importing the source code

Alternatively, the *Fusion* source code from `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/src/fusion_cxx` can be imported into any other project, or compiled into a stand-alone library by the user. This is especially useful if non-standard compiler options should be used.

4.1 Testing the installation and compiling examples

The example directory `<MSKHOME>/mosek/9.3/tools/examples/fusion/cxx` contains a makefile for use with `make` (Linux, Mac OS) or `nmake` (Windows).

To build the examples, open a shell (Linux, Mac OS) or a Developer Command Prompt and go to `<EXDIR>`. To compile all examples run the one of the commands

```
make all  
  
nmake all
```

depending on the operating system. To build only a single example, for instance `lo1.cc`, use one of the following:

```
make lo1  
  
nmake lo1.exe
```

4.2 Creating a Visual Studio Project

The following walk-through describes how to set up a 64bit *Fusion* project with Microsoft Visual Studio 2019. Similar steps should be followed with other tools and setup configurations.

- Create a Visual C++ project or open an existing project in Visual Studio.
- Go to **Project** → **Properties**.
- In the selection box **Configuration:** select **Release** and in **Platform:** select **x64**.
- Go to **Configuration Manager** and set **Active solution configuration** to **Release** and **Active solution platform** to **x64**.
- Under **Configuration Properties** → **C/C++** → **General** → **Additional Include Directories** add the full path to `<HEADERDIR>`.
- Under **Configuration Properties** → **C/C++** → **Code Generation** → **Runtime library** select **Multi-threaded (/MT)**.
- Under **Configuration Properties** → **Linker** → **Input** → **Additional Dependencies** add the full paths to the files `mosek64_9_3.lib` and `fusion64_9_3.lib`.
- To ensure that `mosek64_9_3.dll` is available in the DLL search path during execution, set **Configuration Properties** → **Debugging** → **Environment** to `PATH=%PATH%;<LIBDIR>`.

4.3 Troubleshooting

error LNK2038

The distributed *Fusion* library is built in **StaticRelease** mode. Attempts to link it with an application built in a different mode, like **DynamicDebug** mode, will result in errors such as:

```
error LNK2038: mismatch detected for 'RuntimeLibrary': value 'MT_StaticRelease' doesn  
→ 't match value 'MDd_DynamicDebug' in lo1.obj
```

However, if you must use different configurations for your project, then you may import the *Fusion* source code from `<MSKHOME>/mosek/9.3/tools/platform/<PLATFORM>/src/fusion_cxx` directly into your project and compile within the project with whatever compiler and linker settings you require.

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:

```
auto x = M->variable(n);
auto gamma = M->variable();
auto 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

```
Model::t M = new Model();  auto _M = finally([&]() { M->dispose(); });
```

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

```
auto 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:

```
auto x = M->variable( new_array_ptr<int,1>({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 :

```
auto 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

```
auto 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:

```
auto 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:

```
auto 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( new_array_ptr<Variable::t,1>({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

```
auto A = new_array_ptr<double,2>({
    { 1.0, 2.0, 0.0},
    { 0.0, 1.0, 1.0},
    { 1.0, 0.0, 0.0} });

auto x = M->variable("x",3,Domain::unbounded());
auto 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:

```
auto A = new_array_ptr<double,2>({ {1,2,3,4}, {5,6,7,8} });
auto Ad= Matrix::dense(A);
```

or from a flattened representation:

```
auto Af = new_array_ptr<double,1>({ 1,2,3,4,5,6,7,8 });
auto Aff= Matrix::dense(2,4,Af);
```

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:

```

auto rows    = new_array_ptr<int,1>({ 0, 0, 1, 1 });
auto cols    = new_array_ptr<int,1>({ 0, 3, 1, 3 });
auto values  = new_array_ptr<double,1>({ 1.0, 2.0, 3.0, 4.0 });

auto ms = Matrix::sparse(rows->size(), cols->size(), 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:

```

Variable::t x = M->variable("x", 4);    // Variable

Parameter::t p = M->parameter("p", 4); // Parameter of shape [ 4 ]
Parameter::t 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(new_array_ptr<double,1>({ 1.0, 2.0 , 3.0, 4.0 }));
q->setValue(5.0);

```

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.

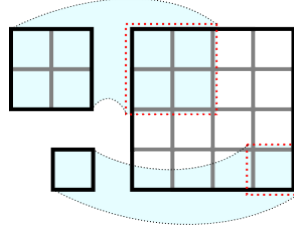


Fig. 6.1: Two dimensional slicing.

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

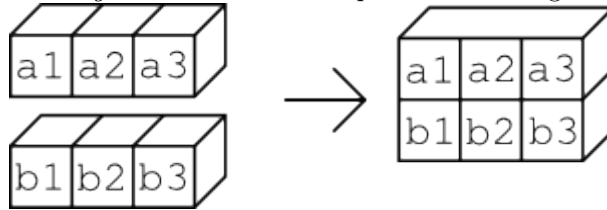
```
auto Axs1 = Ax->slice( new_array_ptr<int,1>({0,0}), new_array_ptr<int,1>({2,2}) );
auto Axs2 = Ax->index( new_array_ptr<int,1>({3,3}) );
```

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 .



```
auto c = Expr::vstack(new_array_ptr<Expression::t,1>({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

$$AX - B = 0.$$

```

auto X = Var::hstack( new_array_ptr<Variable::t,1>({xi(0), xi(1), xi(2), xi(3)})
↳);
auto B = Expr::hstack( new_array_ptr<Expression::t,1>({bi(0), bi(1), bi(2), bi(3)}
↳) );

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:

```

auto 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:

```

double gammaValues[] = {0., 0.5, 1.0};           // Choices for gamma
double beta = 2.0;
auto x = M->variable("x", 1, Domain::greaterThan(0.));
auto y = M->variable("y", 1, Domain::greaterThan(0.));
auto gamma = M->parameter("gamma");

M->objective( ObjectiveSense::Minimize, Expr::add(Expr::mul(gamma, x), Expr::
↳:mul(beta, y)) );

for(auto g : 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.

```
#include "fusion.h"
using namespace mosek::fusion;
using namespace monty;
```

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

```
Model::t M = new Model("lo1"); auto _M = finally([&]() { M->dispose(); });
```

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

```
auto A1 = new_array_ptr<double, 1>({ 3.0, 1.0, 2.0, 0.0 });
auto A2 = new_array_ptr<double, 1>({ 2.0, 1.0, 3.0, 1.0 });
auto A3 = new_array_ptr<double, 1>({ 0.0, 2.0, 0.0, 3.0 });
auto c = new_array_ptr<double, 1>({ 3.0, 1.0, 5.0, 1.0 });
```

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

```
Variable::t 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(A1, x), Domain::equalsTo(30.0));
M->constraint("c2", Expr::dot(A2, x), Domain::greaterThan(15.0));
M->constraint("c3", Expr::dot(A3, 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*.

```

auto sol = x->level();
std::cout << "[x0,x1,x2,x3] = " << (*sol) << "\n";

```

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

```

#include <iostream>
#include "fusion.h"
using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    auto A1 = new_array_ptr<double, 1>({ 3.0, 1.0, 2.0, 0.0 });
    auto A2 = new_array_ptr<double, 1>({ 2.0, 1.0, 3.0, 1.0 });
    auto A3 = new_array_ptr<double, 1>({ 0.0, 2.0, 0.0, 3.0 });
    auto c = new_array_ptr<double, 1>({ 3.0, 1.0, 5.0, 1.0 });

    // Create a model with the name 'lo1'
    Model::t M = new Model("lo1"); auto _M = finally([&]() { M->dispose(); });

    M->setLogHandler([ = ](const std::string & msg) { std::cout << msg << std::flush; }_
    ↪);

    // Create variable 'x' of length 4
    Variable::t 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(A1, x), Domain::equalsTo(30.0));
    M->constraint("c2", Expr::dot(A2, x), Domain::greaterThan(15.0));
    M->constraint("c3", Expr::dot(A3, 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
    auto sol = x->level();
    std::cout << "[x0,x1,x2,x3] = " << (*sol) << "\n";
}

```

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 $\mathcal{K} = \mathcal{K}_1 \times \dots \times \mathcal{K}_l$ 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:

```
Model::t M = new Model("cqo1"); auto _M = finally([&]() { M->dispose(); });
```

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.

```
Variable::t x = M->variable("x", 3, Domain::greaterThan(0.0));
Variable::t 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] ]
Variable::t z1 = Var::vstack(y->index(0), x->slice(0, 2));
Variable::t z2 = Var::vstack(y->slice(1, 3), x->index(2));
```

The linear constraint is defined using the dot product:

```
// Create the constraint
//       $x[0] + x[1] + 2.0 x[2] = 1.0$ 
auto aval = new_array_ptr<double, 1>({1.0, 1.0, 2.0});
M->constraint("lc", Expr::dot(aval, 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.
//       $z1[0] \geq \sqrt{z1[1]^2 + z1[2]^2}$ 
// and  $2.0 z2[0] z2[1] \geq z2[2]^2$ 
Constraint::t qc1 = M->constraint("qc1", z1, Domain::inQCone());
Constraint::t 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();
```

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
ndarray<double, 1> xlv1 = *(x->level());
ndarray<double, 1> ylv1 = *(y->level());
```

```
// Get conic solution of qc1
ndarray<double, 1> qc1lv1 = *(qc1->level());
ndarray<double, 1> qc1dl = *(qc1->dual());
```

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

```
#include <iostream>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    Model::t M = new Model("cqo1"); auto _M = finally([&]() { M->dispose(); });

    Variable::t x = M->variable("x", 3, Domain::greaterThan(0.0));
    Variable::t 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] ]
    Variable::t z1 = Var::vstack(y->index(0), x->slice(0, 2));
    Variable::t z2 = Var::vstack(y->slice(1, 3), x->index(2));
```

(continues on next page)

```

// Create the constraint
//      x[0] + x[1] + 2.0 x[2] = 1.0
auto aval = new_array_ptr<double, 1>({1.0, 1.0, 2.0});
M->constraint("lc", Expr::dot(aval, 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
Constraint::t qc1 = M->constraint("qc1", z1, Domain::inQCone());
Constraint::t qc2 = M->constraint("qc2", z2, Domain::inRotatedQCone());

// Set the objective function to (y[0] + y[1] + y[2])
M->objective("obj", ObjectiveSense::Minimize, Expr::sum(y));

// Solve the problem
M->solve();

// Get the linear solution values
ndarray<double, 1> xlv1 = *(x->level());
ndarray<double, 1> ylv1 = *(y->level());
// Get conic solution of qc1
ndarray<double, 1> qc1lv1 = *(qc1->level());
ndarray<double, 1> qc1dl = *(qc1->dual());

std::cout << "x1,x2,x3 = " << xlv1 << std::endl;
std::cout << "y1,y2,y3 = " << ylv1 << std::endl;
std::cout << "qc1 levels = " << qc1lv1 << std::endl;
std::cout << "qc1 dual conic var levels = " << qc1dl << std::endl;
}

```

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:

```
Model::t M = new Model("pow1"); auto _M = finally([&]() { M->dispose(); });
```

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

```
Variable::t x = M->variable("x", 3, Domain::unbounded());
Variable::t x3 = M->variable();
Variable::t x4 = M->variable();
```

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

```
// Create the linear constraint
auto aval = new_array_ptr<double>, 1>({1.0, 1.0, 0.5});
M->constraint(Expr::dot(x, aval), 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 conic 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:

```
auto cval = new_array_ptr<double>, 1>({1.0, 1.0, -1.0});
M->objective(ObjectiveSense::Maximize, Expr::dot(cval, 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

```
ndarray<double>, 1> xlv1 = *(x->level());
std::cout << "x,y,z = " << xlv1 << std::endl;
```

which is

```
[ 0.06389298  0.78308564  2.30604283 ]
```

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

```
#include <iostream>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    Model::t M = new Model("pow1"); auto _M = finally([&]() { M->dispose(); });

    Variable::t x = M->variable("x", 3, Domain::unbounded());
    Variable::t x3 = M->variable();
    Variable::t x4 = M->variable();

    // Create the linear constraint
    auto aval = new_array_ptr<double>, 1>({1.0, 1.0, 0.5});
    M->constraint(Expr::dot(x, aval), Domain::equalsTo(2.0));

    // Create the conic 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));

    auto cval = new_array_ptr<double>, 1>({1.0, 1.0, -1.0});
    M->objective(ObjectiveSense::Maximize, Expr::dot(cval, Var::vstack(x3, x4, x->
↪index(0))));

    // Solve the problem
    M->solve();

    // Get the linear solution values
    ndarray<double>, 1> xlv1 = *(x->level());
    std::cout << "x,y,z = " << xlv1 << std::endl;
}
```

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 $\mathcal{K} = \mathcal{K}_1 \times \dots \times \mathcal{K}_l$ is a product of smaller cones and x is the full problem variable.

MOSEK can solve conic optimization problems of the form

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

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

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

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:

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

The conic form of (7.5) is:

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

We start by creating the optimization model:

```
Model::t M = new Model("ce01"); auto _M = finally([&]() { M->dispose(); });
```

We then define the variable `x`.

```
Variable::t 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 exponential conic constraint
Constraint::t 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
ndarray<double, 1> xlvl = *(x->level());
```

```
// Get conic solution of expc1
ndarray<double, 1> expclvl = *(expc->level());
ndarray<double, 1> expcdl = *(expc->dual());
```

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

```
#include <iostream>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    Model::t M = new Model("ce01"); auto _M = finally([&]() { M->dispose(); });

    Variable::t 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 exponential conic constraint
    Constraint::t expc = M->constraint("expc", x, Domain::inPExpCone());

    // Set the objective function to (x[0] + x[1])
```

(continues on next page)

```

M->objective("obj", ObjectiveSense::Minimize, Expr::sum(x->slice(0,2)));

// Solve the problem
M->solve();

// Get the linear solution values
ndarray<double, 1> xlv1 = *(x->level());
// Get conic solution of expc1
ndarray<double, 1> expclvl = *(expc->level());
ndarray<double, 1> expcdl = *(expc->dual());

std::cout << "x1,x2,x3 = " << xlv1 << std::endl;
std::cout << "expc levels = " << expclvl << std::endl;
std::cout << "expc dual conic var levels = " << expcdl << std::endl;
}

```

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 \overline{C} , \overline{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:

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

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:

```
Model::t M = new Model("sdo1"); auto _M = finally([&]() { M->dispose(); });
```

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

```
Variable::t X = M->variable("X", Domain::inPSDCone(3));
Variable::t 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).

```
#include <iostream>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    Model::t M = new Model("sdo1"); auto _M = finally([&]() { M->dispose(); });

    // Setting up the variables
    Variable::t X = M->variable("X", Domain::inPSDCone(3));
    Variable::t x = M->variable("x", Domain::inQCone(3));

    // Setting up the constant coefficient matrices
    Matrix::t C = Matrix::dense ( new_array_ptr<double, 2>({{2., 1., 0.}, {1., 2., 1.},
    ↪ {0., 1., 2.}}));
    Matrix::t A1 = Matrix::eye(3);
    Matrix::t 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();

    std::cout << "Solution : " << std::endl;
    std::cout << "  X = " << *(X->level()) << std::endl;
    std::cout << "  x = " << *(x->level()) << std::endl;

    return 0;
}
```

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).

```
#include <iostream>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

std::shared_ptr<ndarray<int,1>> nint(const std::vector<int> &X) { return new_
    ↪array_ptr<int>(X); }
std::shared_ptr<ndarray<double,1>> ndou(const std::vector<double> &X) { return new_
    ↪array_ptr<double>(X); }

int main(int argc, char ** argv)
{
    // Sample data in sparse, symmetric triplet format
    std::vector<int> C1_k = {0, 2};
    std::vector<int> C1_l = {0, 2};
    std::vector<double> C1_v = {1, 6};
    std::vector<int> A1_k = {0, 2, 0, 2};
    std::vector<int> A1_l = {0, 0, 2, 2};
    std::vector<double> A1_v = {1, 1, 1, 2};
    std::vector<int> C2_k = {0, 1, 0, 1, 2};
    std::vector<int> C2_l = {0, 0, 1, 1, 2};
    std::vector<double> C2_v = {1, -3, -3, 2, 1};
    std::vector<int> A2_k = {1, 0, 1, 3};
    std::vector<int> A2_l = {0, 1, 1, 3};
    std::vector<double> A2_v = {1, 1, -1, -3};
    double b = 23;
    double k = -3;

    // Convert input data into Fusion sparse matrices
    auto C1 = Matrix::sparse(3, 3, nint(C1_k), nint(C1_l), ndou(C1_v));
    auto C2 = Matrix::sparse(4, 4, nint(C2_k), nint(C2_l), ndou(C2_v));
    auto A1 = Matrix::sparse(3, 3, nint(A1_k), nint(A1_l), ndou(A1_v));
    auto A2 = Matrix::sparse(4, 4, nint(A2_k), nint(A2_l), ndou(A2_v));

    // Create model
    Model::t M = new Model("sdo2"); auto _M = finally([&]() { M->dispose(); });

    // Two semidefinite variables
    auto X1 = M->variable(Domain::inPSDCone(3));
    auto 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(nint({0,1})), Domain::lessThan(k));
}
```

(continues on next page)

```

// Solve
M->setLogHandler([ = ](const std::string & msg) { std::cout << msg << std::flush;
→} );
M->solve();

// Retrieve solution
std::cout << "Solution (vectorized) : " << std::endl;
std::cout << "  X1 = " << *(X1->level()) << std::endl;
std::cout << "  X2 = " << *(X2->level()) << std::endl;

return 0;
}

```

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.

```
Variable::t 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).

```

#include <iostream>
#include <random>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

// A helper function which returns a slice corresponding to j-th variable
Variable::t slice(Variable::t X, int d, int j) {
    return
        X->slice(new_array_ptr<int,1>({j,0,0}), new_array_ptr<int,1>({j+1,d,d}))
        ->reshape(new_array_ptr<int,1>({d,d}));
}

int main(int argc, char ** argv)
{
    std::random_device rd;
    std::mt19937 e2(rd());
    std::uniform_real_distribution<> dist(0, 1);
}

```

(continues on next page)

```

// Sample data
int n = 100, d = 4, k = 3;
std::vector<double> b({9, 10, 11});
std::vector< std::shared_ptr<ndarray<double,2>> > A;
for(int i=0; i<k*n; i++) {
    auto Ai = std::make_shared<ndarray<double,2>>(shape(d,d));
    for(int s1=0; s1<d; s1++)
        for(int s2=0; s2<=s1; s2++)
            (*Ai)(s1,s2) = (*Ai)(s2,s1) = dist(e2);
    A.push_back(Ai);
}

// Create a model with n semidefinite variables of dimension d x d
Model::t M = new Model("sdo3"); auto _M = finally([&]() { M->dispose(); });

Variable::t X = M->variable(Domain::inPSDCone(d, n));

// Pick indexes (j, s, s), j=0..n-1, s=0..d, of diagonal entries for the
↪ objective
auto alldiag = std::make_shared<ndarray<int,2>>(
    shape(d*n,3),
    std::function<int(const shape_t<2> &)>([d](const shape_t<2> & p) { return
↪ p[1]==0 ? p[0]/d : p[0]%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(int i=0; i<k; i++) {
    std::vector<Expression::t> sumlist;
    for(int j=0; j<n ;j++)
        sumlist.push_back( Expr::dot(A[i*n+j], slice(X, d, j)) );

    M->constraint(Expr::add(new_array_ptr(sumlist)), Domain::greaterThan(b[i]));
}

// Solve
M->setLogHandler([ = ](const std::string & msg) { std::cout << msg << std::flush;
↪ } ); // Add logging
M->writeTask("sdo3.ptf"); // Save problem in readable format
M->solve();

// Get results. Each variable is a slice of X
std::cout << "Contributing variables:" << std::endl;
for(int j=0; j<n; j++) {
    auto Xj = *(slice(X, d, j)->level());
    double maxval = 0;
    for(int s=0; s<d*d; s++) maxval = std::max(maxval, Xj[s]);
    if (maxval>1e-6) {
        std::cout << "X" << j << "=" << std::endl;
        for(int s1=0; s1<d; s1++) {
            for(int s2=0; s2<d; s2++) std::cout << Xj[s1*d+s2] << " ";
            std::cout << std::endl;
        }
    }
}

```

(continues on next page)

```

    }

    return 0;
}

```

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*:

```
Variable::t 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).

```

#include <iostream>
#include <iomanip>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    auto a1 = new_array_ptr<double>, 1>({ 50.0, 31.0 });
    auto a2 = new_array_ptr<double>, 1>({ 3.0, -2.0 });
    auto c = new_array_ptr<double>, 1>({ 1.0, 0.64 });

```

(continues on next page)

```

Model::t M = new Model("milo1"); auto _M = finally([&]() { M->dispose(); });
Variable::t 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(a1, x), Domain::lessThan(250.0));
M->constraint("c2", Expr::dot(a2, x), Domain::greaterThan(-4.0));

// Set max solution time
M->setSolverParam("mioMaxTime", 60.0);
// Set max relative gap (to its default value)
M->setSolverParam("mioTolRelGap", 1e-4);
// Set max absolute gap (to its default value)
M->setSolverParam("mioTolAbsGap", 0.0);

// Set the objective function to (c^T * x)
M->objective("obj", ObjectiveSense::Maximize, Expr::dot(c, x));

// Solve the problem
M->solve();

// Get the solution values
auto sol = x->level();
std::cout << std::setiosflags(std::ios::scientific) << std::setprecision(2)
    << "x1 = " << (*sol)[0] << std::endl
    << "x2 = " << (*sol)[1] << std::endl
    << "MIP rel gap = " << M->getSolverDoubleInfo("mioObjRelGap") << " (" <<
M->getSolverDoubleInfo("mioObjAbsGap") << ")" << std::endl;
}

```

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 x
auto init_sol = new_array_ptr<double>, 1>({ 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

```
int constr = M->getSolverIntInfo("mioConstructSolution");
double constrVal = M->getSolverDoubleInfo("mioConstructSolutionObj");
std::cout << "Initial solution utilization: " << constr << std::endl;
std::cout << "Initial solution objective: " << constrVal << std::endl;
```

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 C++ 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).

```
#include <iostream>
#include <iomanip>
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv)
{
    Model::t M = new Model("mico1"); auto _M = finally([&]() { M->dispose(); });

    Variable::t x = M->variable(Domain::integral(Domain::unbounded()));
    Variable::t y = M->variable(Domain::integral(Domain::unbounded()));
    Variable::t 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->solve();

    std::cout << std::setprecision(2)
```

(continues on next page)

```

    << "x = " << (*(x->level()))[0] << std::endl
    << "y = " << (*(y->level()))[0] << std::endl ;

    return 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

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

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 :

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

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

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

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))) <= 0.
// Each row of [A b] describes one of the exp-terms
void logsumexp(Model::t M,
               std::shared_ptr<ndarray<double, 2>> A,
               Variable::t x,
               std::shared_ptr<ndarray<double, 1>> b)
{
    int k = A->size(0);
    auto 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).

```
std::shared_ptr<ndarray<double, 1>> max_volume_box(double Aw, double Af,
                                                  double alpha, double beta, double
→gamma, double delta)
{
    Model::t M = new Model("max_vol_box"); auto _M = finally([&]() { M->dispose(); });

    auto xyz = M->variable(3);
    M->objective("Objective", ObjectiveSense::Maximize, Expr::sum(xyz));

    logsumexp(M,
              new_array_ptr<double,2>({{1,1,0}, {1,0,1}}),
              xyz,
              new_array_ptr<double,1>({log(2.0/Aw), log(2.0/Aw)}));

    M->constraint(Expr::dot(new_array_ptr<double,1>({0,1,1}), xyz), Domain::
→lessThan(log(Af)));
    M->constraint(Expr::dot(new_array_ptr<double,1>({1,-1,0}), xyz), Domain::
→inRange(log(alpha),log(beta)));
    M->constraint(Expr::dot(new_array_ptr<double,1>({0,-1,1}), xyz), Domain::
→inRange(log(gamma),log(delta)));

    M->setLogHandler([](const std::string & msg) { std::cout << msg << std::flush; });
    M->solve();

    return std::make_shared<ndarray<double, 1>>(shape(3), [xyz](ptrdiff_t i) { return
→exp((*(xyz->level()))[i]); });
}
```

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

Listing 7.15: Sample data for problem (7.16).

```
int main()
{
    double Aw    = 200.0;
    double Af    = 50.0;
    double alpha = 2.0;
    double beta  = 10.0;
    double gamma = 2.0;
    double delta = 10.0;

    auto hwd = max_volume_box(Aw, Af, alpha, beta, gamma, delta);

    std::cout << std::setprecision(4);
    std::cout << "h=" << (*hwd)[0] << " w=" << (*hwd)[1] << " d=" << (*hwd)[2] << std::
    endl;
}
```

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 C++. 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
static void dup(Model::t M, Variable::t x, Variable::t 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
static void abs(Model::t M, Variable::t t, Variable::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 variable
static void norm1(Model::t M, Variable::t t, Variable::t x) {
    Variable::t 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
static void sq(Model::t M, Variable::t t, Variable::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
static void norm2(Model::t M, Variable::t t, Variable::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)
static void pow(Model::t M, Variable::t t, Variable::t x, double p) {
    M->constraint(Expr::hstack(t, 1, x), Domain::inPPowerCone(1.0/p));
}
```

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

Listing 7.22: Power reciprocal.

```
// t >= 1/|x|^p, x>0 (where p>0)
static void pow_inv(Model::t M, Variable::t t, Variable::t x, double 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 variable
static void pnorm(Model::t M, Variable::t t, Variable::t x, double p) {
    int n = (int) x->getSize();
    Variable::t 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 variable of length >= 1
static void geo_mean(Model::t M, Variable::t t, Variable::t x) {
    int n = (int) x->getSize();
    if (n==1) {
        abs(M, x, t);
    }
    else {
        Variable::t t2 = M->variable();
        M->constraint(Expr::hstack(t2, x->index(n-1), t), Domain::inPPowerCone(1.0-1.0/
    ↪n));
        geo_mean(M, t2, x->slice(0,n-1));
    }
}
```

7.8.4 Exponentials and logarithms

log

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

Listing 7.25: Logarithm.

```
// t <= log(x), x>=0
static void log(Model::t M, Variable::t t, Variable::t x) {
    M->constraint(Expr::hstack(x, 1, t), Domain::inPExpCone());
}
```

exp

$$t \geq e^x$$

Listing 7.26: Exponential.

```
// t >= exp(x)
static void exp(Model::t M, Variable::t t, Variable::t x) {
    M->constraint(Expr::hstack(t, 1, x), Domain::inPExpCone());
}
```

Entropy

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

Listing 7.27: Entropy.

```
// t >= x * log(x), x>=0
static void ent(Model::t M, Variable::t t, Variable::t x) {
    M->constraint(Expr::hstack(1, x, Expr::neg(t)), Domain::inPExpCone());
}
```

Relative entropy

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

Listing 7.28: Relative entropy.

```
// t >= x * log(x/y), x,y>=0
static void relent(Model::t M, Variable::t t, Variable::t x, Variable::t 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
static void logsumexp(Model::t M, Variable::t t, Variable::t x) {
    int n = (int) x->getSize();
    Variable::t u = M->variable(n);
    M->constraint(Expr::hstack(u, Expr::constTerm(n, 1.0), Expr::sub(x, Var::repeat(t, n))), Domain::inPExpCone());
    M->constraint(Expr::sum(u), Domain::lessThan(1.0));
}
```

7.8.5 Integer Modeling

Semicontinuous variable

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

Listing 7.30: Semicontinuous variable.

```
// x = 0 or a <= x <= b
static void semicontinuous(Model::t M, Variable::t x, double a, double b) {
    Variable::t 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.
static void indicator(Model::t M, Variable::t t, Variable::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
static void logic_or(Model::t M, Variable::t x, Variable::t y) {
    M->constraint(Expr::add(x, y), Domain::greaterThan(1.0));
}
// x_1 OR ... OR x_n, where x is a binary vector
static void logic_or_vect(Model::t M, Variable::t 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)
static void logic_sos1(Model::t M, Variable::t x) {
    M->constraint(Expr::sum(x), Domain::lessThan(1.0));
}
// NOT(x AND y), where x, y are binary
static void logic_nand(Model::t M, Variable::t x, Variable::t 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.
static void card(Model::t M, Variable::t x, int k) {
    Variable::t 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).

```
static void testExample() {
    Model::t M = new Model();
    Variable::t x = M->variable();
    Variable::t y = M->variable();
    Variable::t 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, new_array_ptr<double>1>({-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.

```
Model::t initializeModel(int m, int n, std::shared_ptr<ndarray<double,2>> A) {
    Model::t M = new Model();
    auto x = M->variable("x", n);

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

    // p_i >= |x_i|, i=1..n
    auto p = M->variable(n);
    M->constraint(Expr::hstack(p, x), Domain::inQCone());

    // q >= |x|_2
    auto q = M->variable();
    M->constraint(Expr::vstack(q, x), Domain::inQCone());

    // Objective, parametrized with lambda1, lambda2
    // t + lambda1*sum(p) + lambda2*q
    auto lambda1 = M->parameter("lambda1");
    auto lambda2 = M->parameter("lambda2");
    auto obj = Expr::add(new_array_ptr<Expression::t, 1>({t, Expr::mul(lambda1, Expr::
    →sum(p)), Expr::mul(lambda2, q)}));
    M->objective(ObjectiveSense::Minimize, obj);

    // 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
int m = 4;
```

(continues on next page)

```

int n = 2;
auto A = new_array_ptr<double, 2>(
    { {1.0, 2.0},
      {3.0, 4.0},
      {-2.0, -1.0},
      {-4.0, -3.0} });

auto M = initializeModel(m, n, A);

// For convenience retrieve some elements of the model
auto b = M->getParameter("b");
auto lambda1 = M->getParameter("lambda1");
auto lambda2 = M->getParameter("lambda2");
auto 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(new_array_ptr<double, 1>({0.1, 1.2, -1.1, 3.0}));
lambda1->setValue(0.1);
lambda2->setValue(0.01);

M->solve();
auto sol = x->level();
std::cout << "Objective " << M->primalObjValue() << ", solution " << (*sol)[0] << ",
↪ " << (*sol)[1] << "\n";

```

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();
sol = x->level();
std::cout << "Objective " << M->primalObjValue() << ", solution " << (*sol)[0] << ",
↪ " << (*sol)[1] << "\n";

```

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(new_array_ptr<double>, 1>({1.0, 1.0, 1.0, 1.0}));
lambda1->setValue(0.0);
lambda2->setValue(0.0);

M->solve();
sol = x->level();
std::cout << "Objective " << M->primalObjValue() << ", solution " << (*sol)[0] << ",
↪ " << (*sol)[1] << "\n";

// And increase lamda2
lambda2->setValue(1.4145);

M->solve();
sol = x->level();
std::cout << "Objective " << M->primalObjValue() << ", solution " << (*sol)[0] << ",
↪ " << (*sol)[1] << "\n";
```

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.cc`, `portfolio_2_frontier.cc`, `portfolio_5_card.cc`, `total_variation.cc`.

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)

```

auto c = new_array_ptr<double, 1>({ 1.5, 2.5, 3.0 });
auto A = new_array_ptr<double, 2>({ {2, 4, 3},
                                     {3, 2, 3},
                                     {2, 3, 2} });
auto b = new_array_ptr<double, 1>({ 100000.0, 50000.0, 60000.0 });
int numvar = 3;
int numcon = 3;

// Create a model and input data
Model::t M = new Model(); auto M_ = monty::finally([&]() { M->dispose(); });

auto x = M->variable(numvar, Domain::greaterThan(0.0));
auto 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.

```
con->index(0)->update(Expr::mul(3.0, x->index(0)), x->index(0));
```

The problem now has the form:

$$\begin{aligned}
 & \text{maximize} && 1.5x_0 + 2.5x_1 + 3.0x_2 \\
 & \text{subject to} && 3x_0 + 4x_1 + 3x_2 \leq 100000, \\
 & && 3x_0 + 2x_1 + 3x_2 \leq 50000, \\
 & && 2x_0 + 3x_1 + 2x_2 \leq 60000,
 \end{aligned} \tag{7.21}$$

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
auto x3 = M->variable(Domain::greaterThan(0.0));
auto xNew = Var::vstack(x, x3);
// Add to the existing constraint
con->update(Expr::mul(x3, new_array_ptr<double, 1>({ 4, 0, 1 })), x3);
// Change the objective to include x3
M->objective(ObjectiveSense::Maximize, Expr::dot(new_array_ptr<double, 1>({1.5, 2.5,
↪3.0, 1.0}), xNew));

```

After this operation the new problem is:

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

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 *****/
auto con2 = M->constraint(Expr::dot(xNew, new_array_ptr<double, 1>({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:

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

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 *****/
// Assemble all constraints we previously defined into one
auto cAll = Constraint::vstack(con, con2);
// Change bounds by effectively updating fixed terms with the difference
cAll->update(new_array_ptr<double, 1>({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	<i>SolutionType.Basic</i>	<i>SolutionType.Interior</i>	
Conic (nonlinear) problem		<i>SolutionType.Interior</i>	
Problem with integer variables			<i>SolutionType.Integer</i>

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.

```
int main(int argc, char** argv)
{
    Model::t M = new Model(); auto _M = finally([&]() { M->dispose(); });

    // (Optional) set a log stream
    // M->setLogHandler([](const std::string & msg) { std::cout << msg << std::flush; }
    ↪);

    // (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
```

(continues on next page)

```

{
    M->solve();

    // We expect solution status OPTIMAL (this is also default)
    M->acceptedSolutionStatus(AccSolutionStatus::Optimal);

    auto x = M->getVariable("x");
    auto xsize = x->getSize();
    auto xVal = x->level();
    std::cout << "Optimal value of x = ";
    for(int i = 0; i < xsize; ++i)
        std::cout << (*xVal)[i] << " ";
    std::cout << "\nOptimal primal objective: " << M->primalObjValue() << "\n";
    // .. continue analyzing the solution

}
catch (const OptimizeError& e)
{
    std::cout << "Optimization failed. Error: " << e.what() << "\n";
}
catch (const SolutionError& e)
{
    // The solution with at least the expected status was not available.
    // We try to diagnose why.
    std::cout << "Requested solution was not available.\n";
    auto prosta = M->getProblemStatus();
    switch(prosta)
    {
        case ProblemStatus::DualInfeasible:
            std::cout << "Dual infeasibility certificate found.\n";
            break;

        case ProblemStatus::PrimalInfeasible:
            std::cout << "Primal infeasibility certificate found.\n";
            break;

        case ProblemStatus::Unknown:
            // The solutions status is unknown. The termination code
            // indicates why the optimizer terminated prematurely.
            std::cout << "The solution status is unknown.\n";
            char symname[MSK_MAX_STR_LEN];
            char desc[MSK_MAX_STR_LEN];
            MSK_getcodedesc((MSKrescodee)(M->getSolverIntInfo("optimizeResponse")), &
→symname, desc);
            std::cout << " Termination code: " << symname << " " << desc << "\n";
            break;

        default:
            std::cout << "Another unexpected problem status: " << prosta << "\n";
    }
}
catch (const std::exception& e)
{
    std::cerr << "Unexpected error: " << e.what() << "\n";
}

```

```

M->dispose();
return 0;
}

```

8.2 Errors and exceptions

Exceptions

Almost every method in Fusion API for C++ 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.

```

try {
    M->setSolverParam("intpntCoTolRelGap", 1.01);
} catch (mosek::fusion::ParameterError& e) {
    std::cout << "Error: " << e.toString() << "\n";
}

```

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.

```

int main(int arc, char** argv)
{
    Model::t M = new Model(); auto _M = finally([&]() { M->dispose(); });

    // (Optional) set a log stream
    // M->setLogHandler([](const std::string & msg) { std::cout << msg << std::flush; }
    ↳);
}

```

(continues on next page)

```

// (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);

    auto x = M->getVariable("x");
    auto xsize = x->getSize();
    auto xVal = x->level();
    std::cout << "Optimal value of x = ";
    for(int i = 0; i < xsize; ++i)
        std::cout << (*xVal)[i] << " ";
    std::cout << "\nOptimal primal objective: " << M->primalObjValue() << "\n";
    // .. continue analyzing the solution
}
catch (const OptimizeError& e)
{
    std::cout << "Optimization failed. Error: " << e.what() << "\n";
}
catch (const SolutionError& e)
{
    // The solution with at least the expected status was not available.
    // We try to diagnose why.
    std::cout << "Requested solution was not available.\n";
    auto prosta = M->getProblemStatus();
    switch(prosta)
    {
        case ProblemStatus::DualInfeasible:
            std::cout << "Dual infeasibility certificate found.\n";
            break;

        case ProblemStatus::PrimalInfeasible:
            std::cout << "Primal infeasibility certificate found.\n";
            break;

        case ProblemStatus::Unknown:
            // The solutions status is unknown. The termination code
            // indicates why the optimizer terminated prematurely.
            std::cout << "The solution status is unknown.\n";
            char symname[MSK_MAX_STR_LEN];
            char desc[MSK_MAX_STR_LEN];
            MSK_getcodedesc((MSKrescodee)(M->getSolverIntInfo("optimizeResponse")), &
↪symname, desc);
            std::cout << " Termination code: " << symname << " " << desc << "\n";
            break;
    }
}

```

```

        default:
            std::cout << "Another unexpected problem status: " << prosta << "\n";
        }
    }
    catch (const std::exception& e)
    {
        std::cerr << "Unexpected error: " << e.what() << "\n";
    }

    M->dispose();
    return 0;
}

```

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( [=](const std::string & msg) { std::cout << msg << std::flush; } );

```

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);
}
catch (mosek::fusion::ParameterError) {
    std::cout << "Wrong parameter value\n";
}
```

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.

```
double tm = M->getSolverDoubleInfo("optimizerTime");
int it = M->getSolverIntInfo("intpntIter");

std::cout << "Time: " << tm << "\nIterations: " << it << "\n";
```

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:

```
bool alive = true;
std::thread T(std::function<void(void)>([&]() { M->solve(); alive = false; }) );
```

In the main thread we are going to check if a time limit has elapsed. 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.

```
bool alive = true;
std::thread T(std::function<void(void)>([&]() { M->solve(); alive = false; }) );

time_t T0 = time(NULL);
while (true)
{
    if (time(NULL) - T0 > timeout)
    {
        std::cout << "Solver terminated due to timeout!\n";
        M->breakSolver();
        T.join();
        break;
    }
    if (! alive)
    {
        std::cout << "Solver terminated before anything happened!\n";
        T.join();
        break;
    }
}
```

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 C++ 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:

```
typedef std::function<bool(MSKcallbackcodee, const double *, const int32_t *, const_
↳int64_t *)> callbackHandler_t;
```

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

```
typedef std::function<int(MSKcallbackcodee)> progressHandler_t;
```

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.

```
static int MSKAPI usercallback( MSKcallbackcodee caller,
                                const double * douinf,
                                const int32_t * intinf,
                                const int64_t * lintinf,
                                Model::t mod,
                                const double maxtime)
{
    switch ( caller )
    {
        case MSK_CALLBACK_BEGIN_INTPNT:
            std::cerr << "Starting interior-point optimizer\n";
            break;
        case MSK_CALLBACK_INTPNT:
            std::cerr << "Iterations: " << intinf[MSK_IINF_INTPNT_ITER];
            std::cerr << " (" << douinf[MSK_DINF_OPTIMIZER_TIME] << "/";
            std::cerr << douinf[MSK_DINF_INTPNT_TIME] << ")s. \n";
            std::cerr << "Primal obj.: " << douinf[MSK_DINF_INTPNT_PRIMAL_OBJ];
            std::cerr << " Dual obj.: " << douinf[MSK_DINF_INTPNT_DUAL_OBJ] << std::endl;
            break;
        case MSK_CALLBACK_END_INTPNT:
            std::cerr << "Interior-point optimizer finished.\n";
            break;
        case MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX:
            std::cerr << "Primal simplex optimizer started.\n";
```

(continues on next page)

```

    break;
case MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX:
    std::cerr << "Iterations: " << intinf[MSK_IINF_SIM_PRIMAL_ITER];
    std::cerr << " Elapsed time: " << douinf[MSK_DINF_OPTIMIZER_TIME];
    std::cerr << "(" << douinf[MSK_DINF_SIM_TIME] << ")\n";
    std::cerr << "Obj.: " << douinf[MSK_DINF_SIM_OBJ] << std::endl;
    break;
case MSK_CALLBACK_END_PRIMAL_SIMPLEX:
    std::cerr << "Primal simplex optimizer finished.\n";
    break;
case MSK_CALLBACK_BEGIN_DUAL_SIMPLEX:
    std::cerr << "Dual simplex optimizer started.\n";
    break;
case MSK_CALLBACK_UPDATE_DUAL_SIMPLEX:
    std::cerr << "Iterations: " << intinf[MSK_IINF_SIM_DUAL_ITER];
    std::cerr << " Elapsed time: " << douinf[MSK_DINF_OPTIMIZER_TIME];
    std::cerr << "(" << douinf[MSK_DINF_SIM_TIME] << ")\n";
    std::cerr << "Obj.: " << douinf[MSK_DINF_SIM_OBJ] << std::endl;
    break;
case MSK_CALLBACK_END_DUAL_SIMPLEX:
    std::cerr << "Dual simplex optimizer finished.\n";
    break;
case MSK_CALLBACK_BEGIN_BI:
    std::cerr << "Basis identification started.\n";
    break;
case MSK_CALLBACK_END_BI:
    std::cerr << "Basis identification finished.\n";
    break;
default:
    break;
}
if ( douinf[MSK_DINF_OPTIMIZER_TIME] >= maxtime )
{
    std::cerr << "MOSEK is spending too much time.Terminate it.\n";
    return 1;
}
return 0;
} /* 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.

```
callbackHandler_t cllbck = [&](MSKcallbackcodee caller,
                             const double * douinf, const int32_t* intinf, const
→int64_t* lintinf)
{
    return usercallback(caller, douinf, intinf, lintinf, M, maxtime);
};

M->setDataCallbackHandler(cllbck);
```

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.

```
#include <fusion.h>
#include <stdlib.h>
#include <iostream>

using namespace mosek::fusion;
using namespace monty;

int main(int argc, char ** argv) {
    if (argc<2) {
        std::cout << "Missing argument, syntax is:" << std::endl;
        std::cout << "  opt_server_sync serveraddr" << std::endl;
        exit(0);
    }
```

(continues on next page)

```

}

std::string addr(argv[1]);

// Setup a simple test problem
Model::t M = new Model("testOptServer"); auto _M = finally([&]() { M->dispose(); }
→ );
Variable::t x = M->variable("x", 3, Domain::greaterThan(0.0));
M->constraint("lc", Expr::dot(new_array_ptr<double, 1>({1.0, 1.0, 2.0}), x),
→ Domain::equalsTo(1.0));
M->objective("obj", ObjectiveSense::Minimize, Expr::sum(x));

// Attach log handler
M->setLogHandler([](const std::string & msg) { std::cout << msg << std::flush; }
→ );

// Set OptServer URL
M->optserverHost(addr);

// Solve the problem on the OptServer
M->solve();

// Get the solution
std::cout << "x1,x2,x3 = " << *(x->level()) << std::endl;

return 0;
}

```

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([=](const std::string & msg) { std::cout << msg << std:::  
↪flush; } );
```

- 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

$$2200 = 200 + 1000 + 1000$$

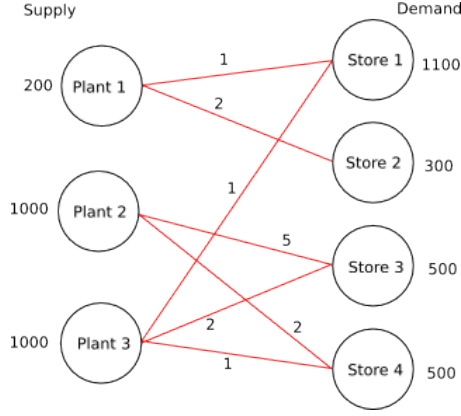


Fig. 9.1: Supply, demand and cost of transportation.

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 C++, 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*.

```
{
    Model::t M = new Model();    auto _M = finally([&]() { M->dispose(); });
    // do something with M
}
```

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
{
    public:
    MyModel(): Model()
    {
        bool finished = false;
        try
        {
            //perform initialization here
            finished = true;
        }
        catch(...)
        {
            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`.

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

```
Expression::t ee = Expr::constTerm(k, 0.0);
for(int i=0;i<n;i++)
    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:

```
auto prods = new ndarray<Expression::t,1>(shape(n));
for(int i=0;i<n;i++) (*prods)[i] = Expr::mul((*A)[i],(*x)[i]);
Expression::t ee = Expr::add( std::shared_ptr<ndarray<Expression::t,1>>(prods) );
```

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 C++ application using the **MOSEK** Fusion API for C++ 9.3.21, the following libraries must be included:

64-bit x86 Linux	64-bit Windows	64-bit Mac OS	Linux ARM64
libmosek64.so.9.3	mosek64_9_3.dll	libmosek64.9.3.dylib	libmosek64.so.9.3
libfusion64.so.9.3	fusion64_9_3.lib	libfusion64.9.3.dylib	libfusion64.so.9.3
libcilkrts.so.5	cilkrts20.dll	libcilkrts.5.dylib	

Chapter 11

Case Studies

In this section we present some case studies in which the Fusion API for C++ 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 C++.

- *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 C++. 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).

```
double BasicMarkowitz
( int                n,
  std::shared_ptr<ndarray<double, 1>> mu,
  std::shared_ptr<ndarray<double, 2>> GT,
  std::shared_ptr<ndarray<double, 1>> x0,
  double             w,
  double             gamma)
{
  Model::t M = new Model("Basic Markowitz"); auto _M = finally([&]() { M->dispose(); }
  ↪);
  // Redirect log output from the solver to stdout for debugging.
  // M->setLogHandler([](const std::string & msg) { std::cout << msg << std::flush; }
  ↪);

  // Defines the variables (holdings). Shortselling is not allowed.
```

(continues on next page)

(continued from previous page)

```
Variable::t x = M->variable("x", n, Domain::greaterThan(0.0));

// Maximize expected return
M->objective("obj", ObjectiveSense::Maximize, Expr::dot(mu, x));

// The amount invested must be identical to initial wealth
M->constraint("budget", Expr::sum(x), Domain::equalsTo(w + sum(x0)));

// Imposes a bound on the risk
M->constraint("risk", Expr::vstack(gamma, Expr::mul(GT, x)), Domain::inQCone());

// Solves the model.
M->solve();

return dot(mu, x->level());
}
```

The source code should be self-explanatory except perhaps for

```
M->constraint("risk", Expr::vstack(gamma, Expr::mul(GT, x)), Domain::inQCone());
```

where the linear expression

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

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.

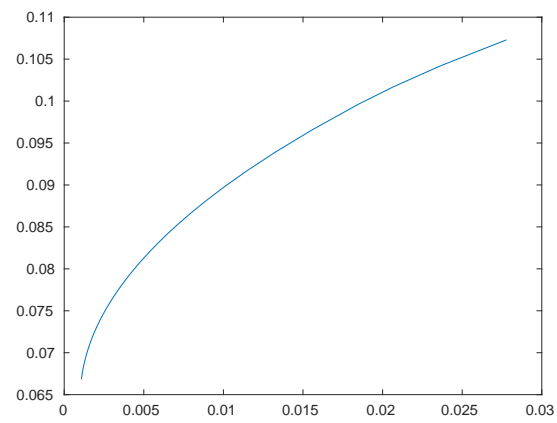


Fig. 11.1: The efficient frontier for the sample data.

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).

```
void EfficientFrontier
( int n,
  std::shared_ptr<ndarray<double, 1>> mu,
  std::shared_ptr<ndarray<double, 2>> GT,
  std::shared_ptr<ndarray<double, 1>> x0,
  double w,
  std::vector<double> & alphas,
  std::vector<double> & frontier_mux,
  std::vector<double> & frontier_s)
{
    Model::t M = new Model("Efficient frontier"); auto M_ = finally([&]() { M->
    dispose(); });

    // Defines the variables (holdings). Shortselling is not allowed.
    Variable::t x = M->variable("x", n, Domain::greaterThan(0.0)); // Portfolio
    variables
    Variable::t s = M->variable("s", 1, Domain::unbounded()); // Variance variable

    M->constraint("budget", Expr::sum(x), Domain::equalsTo(w + sum(x0)));

    // Computes the risk
    M->constraint("variance", Expr::vstack(s, 0.5, Expr::mul(GT, x)), Domain::
    inRotatedQCone());

    // Define objective as a weighted combination of return and variance
    Parameter::t alpha = M->parameter();
    M->objective("obj", ObjectiveSense::Maximize, Expr::sub(Expr::dot(mu, x), Expr::
    mul(alpha, s)));

    // Solve the same problem for many values of parameter alpha
    for (double a : alphas) {
        alpha->setValue(a);
        M->solve();

        frontier_mux.push_back(dot(mu, x->level()));
        frontier_s.push_back((*s->level())[0]);
    }
}
```

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).

```
void MarkowitzWithMarketImpact
( int n,
  std::shared_ptr<ndarray<double, 1>> mu,
  std::shared_ptr<ndarray<double, 2>> GT,
  std::shared_ptr<ndarray<double, 1>> x0,
  double w,
  double gamma,
  std::shared_ptr<ndarray<double, 1>> m,
  std::vector<double> & xsol,
  std::vector<double> & tsol)
{
  Model::t M = new Model("Markowitz portfolio with market impact"); auto M_ =
  ↪finally([&]() { M->dispose(); });

  // Defines the variables. No shortselling is allowed.
  Variable::t x = M->variable("x", n, Domain::greaterThan(0.0));

  // Variables computing the impact cost
  Variable::t t = M->variable("t", n, Domain::unbounded());

  // Maximize expected return
  M->objective("obj", ObjectiveSense::Maximize, Expr::dot(mu, x));

  // Invested amount + slippage cost = initial wealth
  M->constraint("budget", Expr::add(Expr::sum(x), Expr::dot(m, t)), Domain::
  ↪equalsTo(w + sum(x0)));

  // Imposes a bound on the risk
  M->constraint("risk", Expr::vstack( gamma, Expr::mul(GT, x)),
              Domain::inQCone());

  // t >= |x-x0|1.5, using a power cone
  M->constraint("tz", Expr::hstack(t, Expr::constTerm(n, 1.0), Expr::sub(x, x0)),
  ↪Domain::inPPowerCone(2.0/3.0));

  M->solve();

  xsol.resize(n);
  tsol.resize(n);
  auto xlvl = x->level();
  auto tlv1 = t->level();

  std::copy(xlvl->flat_begin(), xlvl->flat_end(), xsol.begin());
  std::copy(tlv1->flat_begin(), tlv1->flat_end(), tsol.begin());
}
```

11.1.6 Transaction Costs

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

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

Hence, whenever asset j is traded we pay a fixed setup cost f_j and a variable cost of g_j per unit traded. Given the assumptions about transaction costs in this section problem (11.6) may be formulated as

$$\begin{aligned} & \text{maximize} && \mu^T x \\ & \text{subject to} && e^T x + f^T y + g^T z = w + e^T x^0, \\ & && (\gamma, G^T x) \in \mathcal{Q}^{n+1}, \\ & && z_j \geq x_j - x_j^0, \quad j = 1, \dots, n, \\ & && z_j \geq x_j^0 - x_j, \quad j = 1, \dots, n, \\ & && z_j \leq U_j y_j, \quad j = 1, \dots, n, \\ & && y_j \in \{0, 1\}, \quad j = 1, \dots, n, \\ & && x \geq 0. \end{aligned} \tag{11.11}$$

First observe that

$$z_j \geq |x_j - x_j^0| = |\Delta x_j|.$$

We choose U_j as some a priori upper bound on the amount of trading in asset j and therefore if $z_j > 0$ then $y_j = 1$ has to be the case. This implies that the transaction cost for asset j is given by

$$f_j y_j + g_j z_j.$$

Example code

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

Listing 11.4: Code solving problem (11.11).

```
std::shared_ptr<ndarray<double, 1>> MarkowitzWithTransactionsCost
( int n,
  std::shared_ptr<ndarray<double, 1>> mu,
  std::shared_ptr<ndarray<double, 2>> GT,
  std::shared_ptr<ndarray<double, 1>> x0,
  double w,
  double gamma,
  std::shared_ptr<ndarray<double, 1>> f,
  std::shared_ptr<ndarray<double, 1>> g)
{
    // Upper bound on the traded amount
    std::shared_ptr<ndarray<double, 1>> u(new ndarray<double, 1>(shape_t<1>(n), w + u
    ↪ sum(x0)));

    Model::t M = new Model("Markowitz portfolio with transaction costs"); auto M_ = u
    ↪ finally([&] () { M->dispose(); });

    // Defines the variables. No shortselling is allowed.
    Variable::t x = M->variable("x", n, Domain::greaterThan(0.0));

    // Additional "helper" variables
    Variable::t z = M->variable("z", n, Domain::unbounded());
    // Binary variables
```

(continues on next page)

```

Variable::t y = M->variable("y", n, Domain::binary());

// Maximize expected return
M->objective("obj", ObjectiveSense::Maximize, Expr::dot(mu, x));

// Invest amount + transactions costs = initial wealth
M->constraint("budget", Expr::add(Expr::add(Expr::sum(x), Expr::dot(f, y)), Expr::
→dot(g, z)),
            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::mul(Matrix::diag(u), y)), Domain::
→lessThan(0.0));

// Integer optimization problems can be very hard to solve so limiting the
// maximum amount of time is a valuable safe guard
M->setSolverParam("mioMaxTime", 180.0);
M->solve();

return x->level();
}

```

11.1.7 Cardinality constraints

Another method to reduce costs involved with processing transactions is to only change positions in a small number of assets. In other words, at most k of the differences $|\Delta x_j| = |x_j - x_j^0|$ are allowed to be non-zero, where k is (much) smaller than the total number of assets n .

This type of constraint can be again modeled by introducing a binary variable y_j which indicates if $\Delta x_j \neq 0$ and bounding the sum of y_j . The basic Markowitz model then gets updated as follows:

$$\begin{aligned}
 &\text{maximize} && \mu^T x \\
 &\text{subject to} && e^T x = w + e^T x^0, \\
 & && (\gamma, G^T x) \in \mathcal{Q}^{n+1}, \\
 & && z_j \geq x_j - x_j^0, & j = 1, \dots, n, \\
 & && z_j \geq x_j^0 - x_j, & j = 1, \dots, n, \\
 & && z_j \leq U_j y_j, & j = 1, \dots, n, \\
 & && y_j \in \{0, 1\}, & j = 1, \dots, n, \\
 & && e^T y \leq k, \\
 & && x \geq 0,
 \end{aligned} \tag{11.12}$$

where U_j is some a priori chosen upper bound on the amount of trading in asset j .

Example code

The following example code demonstrates how to compute an optimal portfolio with cardinality bounds. 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).

```
std::vector<std::vector<double>> MarkowitzWithCardinality
( int n,
  std::shared_ptr<ndarray<double, 1>> mu,
  std::shared_ptr<ndarray<double, 2>> GT,
  std::shared_ptr<ndarray<double, 1>> x0,
  double w,
  double gamma,
  std::vector<int> kValues)
{
    // Upper bound on the traded amount
    std::shared_ptr<ndarray<double, 1>> u(new ndarray<double, 1>(shape_t<1>(n), w +
    ↪sum(x0)));

    Model::t M = new Model("Markowitz portfolio with cardinality constraints"); auto M_
    ↪ = finally([&]() { M->dispose(); });

    // Defines the variables. No shortselling is allowed.
    Variable::t x = M->variable("x", n, Domain::greaterThan(0.0));

    // Additional "helper" variables
    Variable::t z = M->variable("z", n, Domain::unbounded());
    // Binary variables
    Variable::t 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::mul(Matrix::diag(u), y)), Domain::
    ↪lessThan(0.0));

    // At most k assets change position
    auto 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);
```

(continues on next page)

```

// Solve multiple instances by varying the cardinality bound
std::vector<std::vector<double>> results;

for(auto k : kValues) {
    cardMax->setValue(k);
    M->solve();
    auto sol = x->level();
    results.push_back(new_vector_from_array_ptr(sol));
}

return results;
}

```

If we solve our running example with $k = 1, 2, 3$ then we get the following solutions, with increasing expected returns:

Bound: 1	Expected return: 0,0627	Solution: 0,0000 0,0000 1,0000
Bound: 2	Expected return: 0,0669	Solution: 0,0939 0,0000 0,9061
Bound: 3	Expected return: 0,0685	Solution: 0,1010 0,1156 0,7834

11.2 Primal Support-Vector Machine (SVM)

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

The basic SVM model can be stated as:

We are given a set of m points in \mathbb{R}^n , partitioned into two groups. Find, if any, the separating hyperplane of the two subsets with the largest margin, i.e. as far as possible from the points.

Mathematical Model

Let $x_1, \dots, x_m \in \mathbb{R}^n$ be the given training set and let $y_i \in \{-1, +1\}$ be the labels indicating the group membership of the i -th training example. Then we want to determine an affine hyperplane $w^T x = b$ that separates the group in the strong sense that

$$y_i(w^T x_i - b) \geq 1 \quad (11.13)$$

for all i , the property referred to as *large margin classification*: the strip $\{x \in \mathbb{R}^n : -1 < w^T x - b < 1\}$ does not contain any training example. The width of this strip is $2\|w\|^{-1}$, and maximizing that quantity is equivalent to minimizing $\|w\|$. We get that the large margin classification is the solution of the following optimization problem:

$$\begin{aligned} & \text{minimize}_{b,w} && \frac{1}{2}\|w\|^2 \\ & \text{subject to} && y_i(w^T x_i - b) \geq 1 \quad i = 1, \dots, m. \end{aligned}$$

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

To allow more flexibility the soft-margin SVM classifier is often used instead. It admits a violation of the large margin requirement (11.13) by a non-negative slack variable which is then penalized in the objective function.

$$\begin{aligned} & \text{minimize}_{b,w} && \frac{1}{2}\|w\|^2 + C \sum_{i=1}^m \xi_i \\ & \text{subject to} && y_i(w^T x_i - b) \geq 1 - \xi_i \quad i = 1, \dots, m, \\ & && \xi_i \geq 0 \quad i = 1, \dots, m. \end{aligned}$$

In matrix form we have

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

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

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

$$\begin{aligned} & \text{minimize}_{b,w,\xi,t} \quad t + C\mathbf{e}^T\xi \\ & \text{subject to} \quad \xi + y \star (Xw - b\mathbf{e}) \geq \mathbf{e}, \\ & \quad \quad \quad (1, t, w) \in \mathcal{Q}_r^{n+2}, \\ & \quad \quad \quad \xi \geq 0. \end{aligned} \tag{11.14}$$

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

Fusion implementation

We now demonstrate how implement model (11.14). Let us assume that the training examples are stored in the rows of a matrix X , the labels in a vector y and that we have a set of weights C for which we want to train the model. The implementation in *Fusion* of our conic model starts declaring the model class:

```
Model::t M = new Model("primal SVM"); auto _M = finally([&]() { M->dispose(); });
```

Then we proceed defining the variables :

```
Variable::t w = M->variable( n, Domain::unbounded());
Variable::t t = M->variable( 1, Domain::unbounded());
Variable::t b = M->variable( 1, Domain::unbounded());
Variable::t xi = M->variable( 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

```
auto ex = Expr::sub( Expr::mul(X, w), Var::repeat(b, m) );
M->constraint( Expr::add(Expr::mulElm( y, ex ), 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:

```
std::cout << "   c   | b       | w\n";
for (int i = 0; i < nc; i++)
{
    double c = 500.0 * i;
    M->objective( ObjectiveSense::Minimize, Expr::add( t, Expr::mul(c, Expr::sum(xi)
↪) ) );
```

(continues on next page)

```

M->solve();

try
{
    std::cout << std::setw(6) << c << " | " << std::setw(8) << (*(b->level())) [0] <
↪< " | ";
    std::cout.width(8);
    auto wlev = w->level();
    std::copy( wlev->begin(), wlev->end() , std::ostream_iterator<double>(std::cout,
↪ " ") );
    std::cout << "\n";
}
catch (...) {}

}

```

Source code

Listing 11.6: The code implementing model (11.14)

```

Model::t M = new Model("primal SVM"); auto _M = finally([&]() { M->dispose(); });

Variable::t w = M->variable( n, Domain::unbounded());
Variable::t t = M->variable( 1, Domain::unbounded());
Variable::t b = M->variable( 1, Domain::unbounded());
Variable::t xi = M->variable( m, Domain::greaterThan(0.));

auto ex = Expr::sub( Expr::mul(X, w), Var::repeat(b, m) );
M->constraint( Expr::add(Expr::mulElm( y, ex ), xi ), Domain::greaterThan( 1. ) );

M->constraint( Expr::vstack(1., t, w) , Domain::inRotatedQCone() );

std::cout << "    c    | b        | w\n";
for (int i = 0; i < nc; i++)
{
    double c = 500.0 * i;
    M->objective( ObjectiveSense::Minimize, Expr::add( t, Expr::mul(c, Expr::sum(xi)
↪) ) );
    M->solve();

    try
    {
        std::cout << std::setw(6) << c << " | " << std::setw(8) << (*(b->level())) [0] <
↪< " | ";
        std::cout.width(8);
        auto wlev = w->level();
        std::copy( wlev->begin(), wlev->end() , std::ostream_iterator<double>(std::cout,
↪ " ") );
        std::cout << "\n";
    }
    catch (...) {}
}

```

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.

```
int m = 50 ;
int n = 3;
int nc = 10;

int nump = m / 2;
int numm = m - nump;

auto y = new_array_ptr<double, 1> (m);
std::fill( y->begin(), y->begin() + nump, 1.);
std::fill( y->begin() + nump, y->end(), -1.);

double mean = 1.;
double var = 1.;

auto X = std::shared_ptr< ndarray<double, 2> > ( new ndarray<double, 2> ( shape_t
-><2>(m, n) ) );

std::mt19937 e2(0);

for (int i = 0; i < nump; i++)
{
    auto ram = std::bind(std::normal_distribution<>(mean, var), e2);
    for ( int j = 0; j < n; j++)
        (*X)(i, j) = ram();
}

std::cout << "Number of data      : " << m << std::endl;
std::cout << "Number of features: " << n << std::endl;
```

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.

11.3 2D Total Variation

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

Mathematical Formulation

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

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

where $u_{ij} \in [0, 1]$ is the actual signal value and v_{ij} is the noise. The aim is to reconstruct u subtracting the noise from the observations.

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

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

which translates into a simple conic quadratic constraint as

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

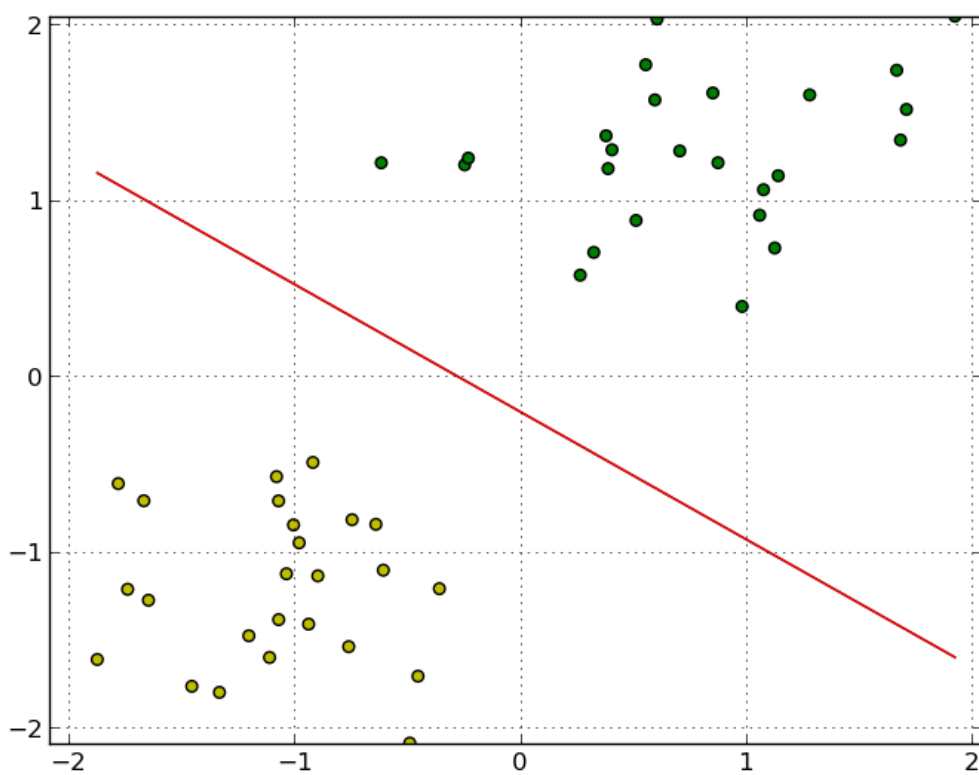


Fig. 11.2: Separating hyperplane for two clusters of points.

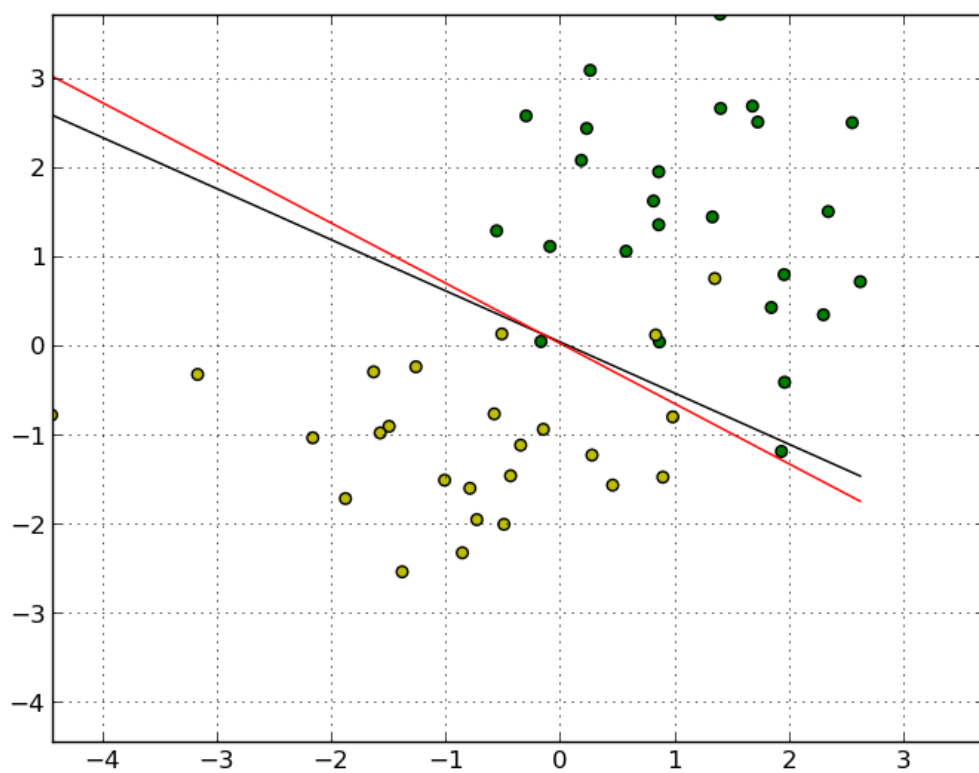


Fig. 11.3: Soft separating hyperplane for two groups of points.

We aim to minimize the change in signal value when moving between adjacent cells. To this end we define the adjacent differences vector as

$$\partial_{ij}^+ = \begin{pmatrix} \partial_{ij}^x \\ \partial_{ij}^y \end{pmatrix} = \begin{pmatrix} u_{i+1,j} - u_{i,j} \\ u_{i,j+1} - u_{i,j} \end{pmatrix}, \quad (11.15)$$

for each cell $1 \leq i, j \leq n$ (we assume that the respective coordinates ∂_{ij}^x and ∂_{ij}^y are zero on the right and bottom boundary of the grid).

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

$$t_{ij} \geq \|\partial_{ij}^+\|_2 \quad \text{or} \quad (t_{ij}, \partial_{ij}^+) \in \mathcal{Q},$$

and minimize the sum of all t_{ij} .

The complete model takes the form:

$$\begin{aligned} \min \quad & \sum_{1 \leq i, j \leq n} t_{ij}, \\ \text{s.t.} \quad & \partial_{ij}^+ = (u_{i+1,j} - u_{i,j}, u_{i,j+1} - u_{i,j})^T, \quad \forall 1 \leq i, j \leq n, \\ & (t_{ij}, \partial_{ij}^+) \in \mathcal{Q}^3, \quad \forall 1 \leq i, j \leq n, \\ & (\sigma, u - f) \in \mathcal{Q}^{nm+1}, \\ & u_{i,j} \in [0, 1]. \quad \forall 1 \leq i, j \leq n. \end{aligned} \quad (11.16)$$

Implementation

The *Fusion* implementation of model (11.16) uses variable and expression slices.

First of all we start by creating the optimization model and variables \mathbf{t} and \mathbf{u} . Since we intend to solve the problem many times with various input data we define σ and f as parameters:

```
Model::t M = new Model("TV");

Variable::t u = M->variable("u", new_array_ptr<int, 1>({n + 1, m + 1}), Domain::
↳ inRange(0.0, 1.0));
Variable::t t = M->variable("t", new_array_ptr<int, 1>({n, m}), Domain::
↳ unbounded());

// In this example we define sigma and the input image f as parameters
// to demonstrate how to solve the same model with many data variants.
// Of course they could simply be passed as ordinary arrays if that is not needed.
Parameter::t sigma = M->parameter("sigma");
Parameter::t f = M->parameter("f", n, m);
```

Note the dimensions of \mathbf{u} is larger than those of the grid to accommodate the boundary conditions later. The actual cells of the grid are defined as a slice of \mathbf{u} :

```
Variable::t ucore = u->slice(new_array_ptr<int, 1>({0, 0}), new_array_ptr<int, 1>({n, m}));
```

The next step is to define the partial variation along each axis, as in (11.15):

```
Expression::t deltax = Expr::sub( u->slice( new_array_ptr<int, 1>({1, 0}), new_
↳ array_ptr<int, 1>({n + 1, m}) ), ucore );
Expression::t deltay = Expr::sub( u->slice( new_array_ptr<int, 1>({0, 1}), new_
↳ array_ptr<int, 1>({n, m + 1}) ), ucore );
```

Slices are created on the fly as they will not be reused. Now we can set the conic constraints on the norm of the total variations. To this extent we stack the variables \mathbf{t} , \mathbf{deltax} and \mathbf{deltay} together and demand that each row of the new matrix is in a quadratic cone.

```
M->constraint( Expr::stack(2, t, deltax, deltax), Domain::inQCone()->axis(2) );
```

We now need to bound the norm of the noise. This can be achieved with a conic constraint using `f` as a one-dimensional array:

```
M->constraint( Expr::vstack(sigma, Expr::flatten( Expr::sub(f, ucore) ) ),
              Domain::inQCone() );
```

The objective function is the sum of all t_{ij} :

```
M->objective( ObjectiveSense::Minimize, Expr::sum(t) );
```

Example

Consider the linear signal $u_{ij} = \frac{i+j}{n+m}$ and its modification with random Gaussian noise, as in Fig. 11.4. Various reconstructions of u , obtained with different values of σ , are shown in Fig. 11.5 (where $\bar{\sigma} = \sigma/nm$ is the relative noise bound per cell).

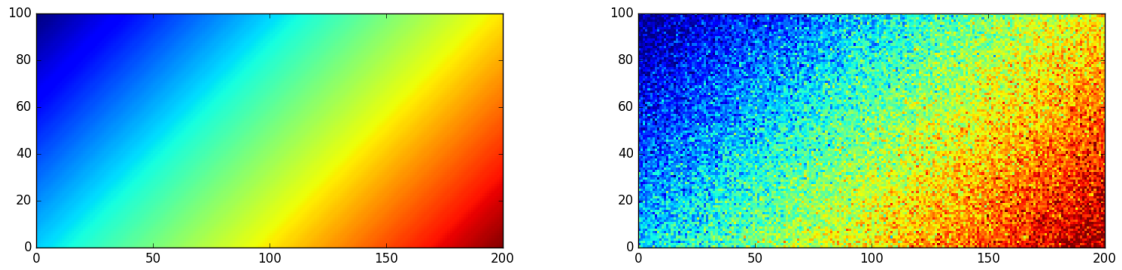


Fig. 11.4: A linear signal and its modification with random Gaussian noise.

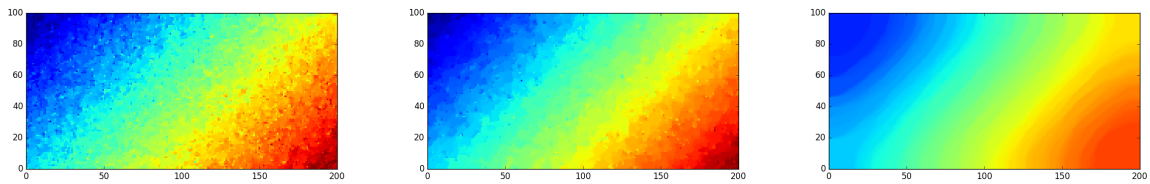


Fig. 11.5: Three reconstructions of the linear signal obtained for $\bar{\sigma} \in \{0.0004, 0.0005, 0.0006\}$, respectively.

Source code

Listing 11.7: The *Fusion* implementation of model (11.16).

```

Model::t total_var(int n, int m) {
    Model::t M = new Model("TV");

    Variable::t u = M->variable("u", new_array_ptr<int, 1>({n + 1, m + 1}), Domain::
    ↳inRange(0.0, 1.0));
    Variable::t t = M->variable("t", new_array_ptr<int, 1>({n, m}), Domain::
    ↳unbounded());

    // In this example we define sigma and the input image f as parameters
    // to demonstrate how to solve the same model with many data variants.
    // Of course they could simply be passed as ordinary arrays if that is not needed.
    Parameter::t sigma = M->parameter("sigma");
    Parameter::t f = M->parameter("f", n, m);

    Variable::t ucore = u->slice(new_array_ptr<int, 1>({0, 0}), new_array_ptr<int, 1>({
    ↳n, m}));

    Expression::t deltax = Expr::sub( u->slice( new_array_ptr<int, 1>({1, 0}), new_
    ↳array_ptr<int, 1>({n + 1, m}) ), ucore );
    Expression::t deltax = Expr::sub( u->slice( new_array_ptr<int, 1>({0, 1}), new_
    ↳array_ptr<int, 1>({n, m + 1}) ), ucore );

    M->constraint( Expr::stack(2, t, deltax, deltax), Domain::inQCone()->axis(2) );

    M->constraint( Expr::vstack(sigma, Expr::flatten( Expr::sub(f, ucore) ) ),
    ↳Domain::inQCone() );

    M->objective( ObjectiveSense::Minimize, Expr::sum(t) );

    return M;
}

int main(int argc, char ** argv)
{
    std::normal_distribution<double> ndistr(0., 1.);
    std::mt19937 engine(0);

    int n = 100;
    int m = 200;
    std::vector<double> sigmas({ 0.0004, 0.0005, 0.0006 });

    // Create a parametrized model with given shape
    Model::t M = total_var(n, m);
    Parameter::t sigma = M->getParameter("sigma");
    Parameter::t f = M->getParameter("f");
    Variable::t ucore = M->getVariable("u")->slice(new_array_ptr<int, 1>({0, 0}), new_
    ↳array_ptr<int, 1>({n, m}));

    // Example: Linear signal with Gaussian noise
    std::vector<std::vector<double>> signal(n, std::vector<double>(m));
    std::vector<std::vector<double>> noise(n, std::vector<double>(m));
    std::vector<std::vector<double>> fVal(n, std::vector<double>(m));
    std::vector<std::vector<double>> sol(n, std::vector<double>(m));

```

(continues on next page)

```

for(int i=0; i<n; i++) for(int j=0; j<m; j++) {
    signal[i][j] = 1.0*(i+j)/(n+m);
    noise[i][j] = ndistr(engine) * 0.08;
    fVal[i][j] = std::max( std::min(1.0, signal[i][j] + noise[i][j]), .0 );
}

// Set value for f
f->setValue(new_array_ptr(fVal));

for(int iter=0; iter<3; iter++) {
    // Set new value for sigma and solve
    sigma->setValue(sigmas[iter]*n*m);

    M->solve();

    // Retrieve solution from ucore
    auto ucoreLev = *(ucore->level());
    for(int i=0; i<n; i++) for(int j=0; j<m; j++)
        sol[i][j] = ucoreLev[i*n+m];

    // Use the solution
    // ...

    std::cout << "rel_sigma = " << sigmas[iter] << " total_var = " << M->
    primalObjValue() << std::endl;
}

M->dispose();

return 0;
}

```

11.4 Multiprocessor Scheduling

In this case study we consider a simple scheduling problem in which a set of jobs must be assigned to a set of identical machines. We want to minimize the makespan of the overall processing, i.e. the latest machine termination time.

The main aims of this case study are

- to show how to define a Integer Linear Programming model,
- to take advantage of *Fusion* operators to compactly express sets of constraints,
- to provide the solver with an incumbent integer feasible solution.

Mathematical formulation

We are given a set of jobs J with $|J| = n$ to be assigned to a set M of identical machines with $|M| = m$. Each job $j \in J$ has a processing time $T_j > 0$ and can be assigned to any machine. Our aim is to find the job scheduling that minimizes the overall makespan, i.e. the maximum completion time among all machines.

Formally, we introduce a binary variable x_{ij} that takes value 1 if the job j is assigned to the machine i , zero otherwise. The only constraint we need to set is the requirement that a job must be assigned to a single machine. The optimization model takes the following form:

$$\begin{aligned} & \min \max_{i \in M} \sum_{j \in J} T_j x_{ij} \\ \text{s.t. } & \sum_{i \in M} x_{ij} = 1, & j \in J, \\ & x_{ij} \in \{0, 1\} & \forall i \in M, j \in J. \end{aligned} \quad (11.17)$$

Model (11.17) can be easily transformed into an integer linear programming model as follows:

$$\begin{aligned} & \min t \\ \text{s.t. } & \sum_{i \in M} x_{ij} = 1, & j \in J, \\ & t \geq \sum_{j \in J} T_j x_{ij}, & i \in M, \\ & x_{ij} \in \{0, 1\}, & \forall i \in M, j \in J. \end{aligned} \quad (11.18)$$

The implementation of this model in *Fusion* is straightforward:

```
Model::t M = new Model("Multi-processor scheduling"); auto _M = finally([&]() { M->
↳dispose(); });

Variable::t x = M->variable("x", new_array_ptr<int, 1>({m, n}), Domain::binary());
Variable::t t = M->variable("t", 1, Domain::unbounded());

M->constraint( Expr::sum(x, 0), Domain::equalsTo(1.) );
M->constraint( Expr::sub( Var::repeat(t, m), Expr::mul(x, T) ), Domain::
↳greaterThan(0.) );

M->objective( ObjectiveSense::Minimize, t );
```

Most of the code is self-explanatory. The only critical point is

```
M->constraint( Expr::sub( Var::repeat(t, m), Expr::mul(x, T) ), Domain::
↳greaterThan(0.) );
```

that implements the set of constraints

$$t \geq \sum_{j \in J} T_j x_{ij}, \quad i \in M.$$

To fit in *Fusion* we restate the constraints as

$$t - \sum_{j \in J} T_j x_{ij} \geq 0, \quad i \in M,$$

which corresponds in matrix form to

$$t\mathbf{1} - xT \geq 0. \quad (11.19)$$

The function *Var.repeat* creates a vector of length m , as required for (11.19). The same result can be obtained via matrix multiplication, i.e. using *Expr.mul*, but in this particular case *Var.repeat* is faster as it only performs a logical operation.

Longest Processing Time first rule (LPT)

The multiprocessor scheduling is known to be an NP-complete problem (see [GJ79]). Nevertheless there are effective heuristics, with provable worst case bounds, that are able to provide a good integer solution quickly. In particular, we will use the so-called *Longest Processing Time first* rule (LPT, proposed in [Gra69]).

The informal algorithm sketch is the following:

- while M is not empty do
 - let k be the machine with the smallest load so far,
 - let i be the job in M with the longest completion time,
 - assign job i to machine k ,
 - update the load of machine k ,
 - remove i from M .

This simple algorithm is a $\frac{1}{3}(4 - \frac{1}{m})$ approximation. So for $m = 1$ we get the optimal solution (indeed there is no choice with a single machine); for $m \rightarrow \infty$ the approximation factor is no worse than $4/3$ (again see [Gra69]).

A simple implementation is given below.

```
//LPT heuristic
auto schedule = std::shared_ptr<ndarray<double, 1>>(new ndarray<double, 1>(m, 0.));
auto init = std::shared_ptr<ndarray<double, 1>>(new ndarray<double, 1>(n * m, 0.));

for (int i = 0; i < n; i++)
{
    auto pos = std::distance(schedule->begin(), std::min_element(schedule->begin(),
↪schedule->end()));
    (*schedule)[pos] += (*T)[i];
    (*init)[pos * n + i] = 1;
}
```

An efficient implementation of the LPT rule is beyond the scope of this section. The important part is that the scheduling produced by the LPT algorithm can be used as incumbent solution for the **MOSEK** mixed-integer linear programming solver. The availability of an integer feasible solution can significantly improve the performance of the solver.

To input the solution we only need to use the `Variable.setLevel` method, as shown below

```
x->setLevel(init);
```

We can test the program with and without providing the initial LPT solution. Our random datasets consists of a mix of tasks with long and short processing times and we accept a solution at relative optimality tolerance 0.01. Some results are shown in the table below.

Table 11.1: Sample test results for the makespan problem.

n	m	long tasks	short tasks	No LPT	With LPT
1000	8	20%	80%	13.36s	1.23s
1000	8	80%	20%	1.35s	1.24s
100	12	20%	80%	16.37s	0.11s
100	12	80%	20%	16.62s	10.01s
20	20	0%	100%	10.38s	21.88s

We can see that depending on the structure and parameters of the problem it may pay off to provide an initial LPT solution. Therefore it is always recommended to test the mixed-integer solver with different settings to find the most efficient setup for a given problem.

Listing 11.8: Complete code for the LPT scheduling example.

```
#include <iostream>
#include <random>
#include <sstream>

#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

int main(int arc, char** argv)
{
    double lb = 1.0;           //Bounds for the length of a short task
    double ub = 5.;

    int n = 30;                //Number of tasks
    int m = 6;                  //Number of processors

    double sh = 0.8;           //The proportion of short tasks
    int n_short = (int)(sh * n);
    int n_long = n - n_short;

    auto gen = std::bind(std::uniform_real_distribution<double>(lb, ub), std::mt19937(0));

    auto T = std::shared_ptr<ndarray<double, 1>>(new ndarray<double, 1>(n));
    for (int i = 0; i < n_short; i++) (*T)[i] = gen();
    for (int i = n_short; i < n; i++) (*T)[i] = 20 * gen();
    std::sort(T->begin(), T->end(), std::greater<double>());

    Model::t M = new Model("Multi-processor scheduling"); auto _M = finally([&]() { M->dispose(); });

    Variable::t x = M->variable("x", new_array_ptr<int, 1>({m, n}), Domain::binary());
    Variable::t 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
    auto schedule = std::shared_ptr<ndarray<double, 1>>(new ndarray<double, 1>(m, 0.));
    auto init = std::shared_ptr<ndarray<double, 1>>(new ndarray<double, 1>(n * m, 0.));

    for (int i = 0; i < n; i++)
    {
        auto pos = std::distance(schedule->begin(), std::min_element(schedule->begin(),
        schedule->end()));
        (*schedule)[pos] += (*T)[i];
        (*init)[pos * n + i] = 1;
    }

    //Comment this line to switch off feeding in the initial LPT solution
}
```

(continues on next page)

```

x->setLevel(init);

M->setLogHandler([ = ](const std::string & msg) { std::cout << msg << std::flush; }
↪);

M->setSolverParam("mioTolRelGap", .01);
M->solve();

std::cout << "initial solution: \n";
for (int i = 0; i < m; i++)
{
    std::cout << "M " << i << " [";
    for (int y = 0; y < n; y++)
        std::cout << int( (*init)[i * n + y] ) << ", ";
    std::cout << "]\n";
}

std::cout << "MOSEK solution:\n";
for (int i = 0; i < m; i++)
{
    std::cout << "M " << i << " [";
    for (int y = 0; y < n; y++)
        std::cout << int((*(x->index(i, y)->level()))[0]) << ", ";
    std::cout << "]\n";
}

return 0;
}

```

11.5 Logistic regression

Logistic regression is an example of a binary classifier, where the output takes one two values 0 or 1 for each data point. We call the two values *classes*.

Formulation as an optimization problem

Define the sigmoid function

$$S(x) = \frac{1}{1 + \exp(-x)}.$$

Next, given an observation $x \in \mathbb{R}^d$ and a weights $\theta \in \mathbb{R}^d$ we set

$$h_{\theta}(x) = S(\theta^T x) = \frac{1}{1 + \exp(-\theta^T x)}.$$

The weights vector θ is part of the setup of the classifier. The expression $h_{\theta}(x)$ is interpreted as the probability that x belongs to class 1. When asked to classify x the returned answer is

$$x \mapsto \begin{cases} 1 & h_{\theta}(x) \geq 1/2, \\ 0 & h_{\theta}(x) < 1/2. \end{cases}$$

When training a logistic regression algorithm we are given a sequence of training examples x_i , each labelled with its class $y_i \in \{0, 1\}$ and we seek to find the weights θ which maximize the likelihood function

$$\prod_i h_{\theta}(x_i)^{y_i} (1 - h_{\theta}(x_i))^{1-y_i}.$$

Of course every single y_i equals 0 or 1, so just one factor appears in the product for each training data point. By taking logarithms we can define the logistic loss function:

$$J(\theta) = - \sum_{i:y_i=1} \log(h_\theta(x_i)) - \sum_{i:y_i=0} \log(1 - h_\theta(x_i)).$$

The training problem with regularization (a standard technique to prevent overfitting) is now equivalent to

$$\min_{\theta} J(\theta) + \lambda \|\theta\|_2.$$

This can equivalently be phrased as

$$\begin{aligned} & \text{minimize} && \sum_i t_i + \lambda r \\ & \text{subject to} && \begin{aligned} t_i &\geq -\log(h_\theta(x)) &= \log(1 + \exp(-\theta^T x_i)) & \text{if } y_i = 1, \\ t_i &\geq -\log(1 - h_\theta(x)) &= \log(1 + \exp(\theta^T x_i)) & \text{if } y_i = 0, \\ r &\geq \|\theta\|_2. \end{aligned} \end{aligned} \quad (11.20)$$

Implementation

As can be seen from (11.20) the key point is to implement the softplus bound $t \geq \log(1 + e^u)$, which is the simplest example of a log-sum-exp constraint for two scalar variables t, u . This is equivalent to

$$\exp(u - t) + \exp(-t) \leq 1$$

and further to

$$\begin{aligned} (z_1, 1, u - t) &\in K_{\text{exp}} & (z_1 \geq \exp(u - t)), \\ (z_2, 1, -t) &\in K_{\text{exp}} & (z_2 \geq \exp(-t)), \\ z_1 + z_2 &\leq 1. \end{aligned} \quad (11.21)$$

Listing 11.9: Implementation of $t \geq \log(1 + e^u)$ as in (11.21).

```
// t >= log( 1 + exp(u) ) coordinatewise
void softplus(Model::t M,
              Expression::t t,
              Expression::t u)
{
    int n = (*t->getShape())[0];
    auto z1 = M->variable(n);
    auto z2 = M->variable(n);
    M->constraint(Expr::add(z1, z2), Domain::equalsTo(1));
    M->constraint(Expr::hstack(z1, Expr::constTerm(n, 1.0), Expr::sub(u,t)), Domain::
    inPEXPcone());
    M->constraint(Expr::hstack(z2, Expr::constTerm(n, 1.0), Expr::neg(t)), Domain::
    inPEXPcone());
}
```

Once we have this subroutine, it is easy to implement a function that builds the regularized loss function model (11.20).

Listing 11.10: Implementation of (11.20).

```
// Model logistic regression (regularized with full 2-norm of theta)
// X - n x d matrix of data points
// y - length n vector classifying training points
// lamb - regularization parameter
std::pair<Model::t, Variable::t>
logisticRegression(std::vector<std::vector<double>> & X,
```

(continues on next page)

```

        std::vector<bool>          & y,
        double                    lamb)
{
    int n = X.size();
    int d = X[0].size();          // num samples, dimension

    Model::t M = new Model();

    auto theta = M->variable(d);
    auto t      = M->variable(n);
    auto reg     = M->variable();

    M->objective(ObjectiveSense::Minimize, Expr::add(Expr::sum(t), Expr::mul(lamb,
    ↪reg)));
    M->constraint(Var::vstack(reg, theta), Domain::inQCone());

    auto signs = std::make_shared<ndarray<double,1>>(shape(n), [y](ptrdiff_t i) {
    ↪return y[i] ? -1 : 1; });

    softplus(M, t, Expr::mulElm(Expr::mul(new_array_ptr<double>(X), theta), signs));

    return std::make_pair(M, theta);
}

```

Example: 2D dataset fitting

In the next figure we apply logistic regression to the training set of 2D points taken from the example `ex2data2.txt`. The two-dimensional dataset was converted into a feature vector $x \in \mathbb{R}^{28}$ using monomial coordinates of degrees at most 6.

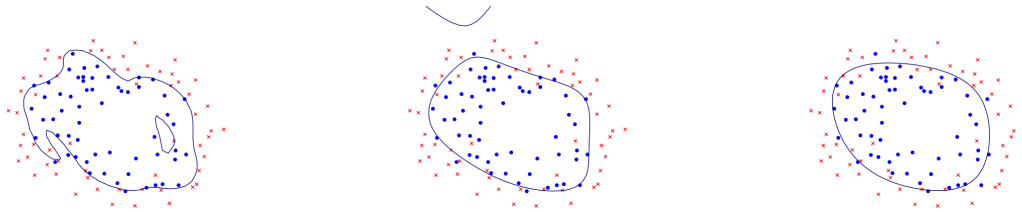


Fig. 11.6: Logistic regression example with none, medium and strong regularization (small, medium, large λ). Without regularization we get obvious overfitting.

11.6 Inner and outer Löwner-John Ellipsoids

In this section we show how to compute the Löwner-John *inner* and *outer* ellipsoidal approximations of a polytope. They are defined as, respectively, the largest volume ellipsoid contained inside the polytope and the smallest volume ellipsoid containing the polytope, as seen in Fig. 11.7.

For further mathematical details, such as uniqueness of the two ellipsoids, consult [BenTalN01]. Our solution is a mix of conic quadratic and semidefinite programming. Among other things, in Sec. 11.6.3 we show how to implement bounds involving the determinant of a PSD matrix.

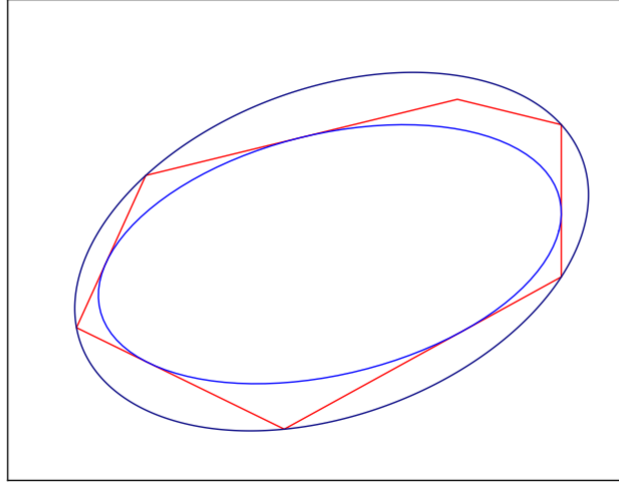


Fig. 11.7: The inner and outer Löwner-John ellipse of a polygon.

11.6.1 Inner Löwner-John Ellipsoids

Suppose we have a polytope given by an h-representation

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

and we wish to find the inscribed ellipsoid with maximal volume. It will be convenient to parametrize the ellipsoid as an affine transformation of the standard disk:

$$\mathcal{E} = \{x \mid x = Cu + d, u \in \mathbb{R}^n, \|u\|_2 \leq 1\}.$$

Every non-degenerate ellipsoid has a parametrization such that C is a positive definite symmetric $n \times n$ matrix. Now the volume of \mathcal{E} is proportional to $\det(C)^{1/n}$. The condition $\mathcal{E} \subseteq \mathcal{P}$ is equivalent to the inequality $A(Cu + d) \leq b$ for all u with $\|u\|_2 \leq 1$. After a short computation we obtain the formulation:

$$\begin{aligned} & \text{maximize} && t \\ & \text{subject to} && t \leq \det(C)^{1/n}, \\ & && (b - Ad)_i \geq \|(AC)_i\|_2, \quad i = 1, \dots, m, \\ & && C \succeq 0, \end{aligned} \tag{11.22}$$

where X_i denotes the i -th row of the matrix X . This can easily be implemented using *Fusion*, where the sequence of conic inequalities can be realized at once by feeding in the matrices $b - Ad$ and AC .

Listing 11.11: *Fusion* implementation of model (11.22).

```
std::pair<std::shared_ptr<ndarray<double, 1>>, std::shared_ptr<ndarray<double, 1>>>
  lowerjohn_inner
  ( std::shared_ptr<ndarray<double, 2>> A,
    std::shared_ptr<ndarray<double, 1>> b)
{
  Model::t M = new Model("lowerjohn_inner"); auto _M = finally([&]() { M->dispose();
  ↪});
  int m = A->size(0);
  int n = A->size(1);

  // Setup variables
  Variable::t t = M->variable("t", 1, Domain::greaterThan(0.0));
```

(continues on next page)

(continued from previous page)

```
Variable::t C = det_rootn(M, t, n);
Variable::t d = M->variable("d", n, Domain::unbounded());

// quadratic cones
M->constraint(Expr::hstack(Expr::sub(b, Expr::mul(A, d)), Expr::mul(A, C)),
             Domain::inQCone());

// Objective: Maximize t
M->objective(ObjectiveSense::Maximize, t);
M->solve();

return std::make_pair(C->level(), d->level());
}
```

The only black box is the method `det_rootn` which implements the constraint $t \leq \det(C)^{1/n}$. It will be described in Sec. 11.6.3.

11.6.2 Outer Löwner-John Ellipsoids

To compute the outer ellipsoidal approximation to a polytope, let us now start with a v-representation

$$\mathcal{P} = \text{conv}\{x_1, x_2, \dots, x_m\} \subseteq \mathbb{R}^n,$$

of the polytope as a convex hull of a set of points. We are looking for an ellipsoid given by a quadratic inequality

$$\mathcal{E} = \{x \in \mathbb{R}^n \mid \|Px - c\|_2 \leq 1\},$$

whose volume is proportional to $\det(P)^{-1/n}$, so we are after maximizing $\det(P)^{1/n}$. Again, there is always such a representation with a symmetric, positive definite matrix P . The inclusion conditions $x_i \in \mathcal{E}$ translate into a straightforward problem formulation:

$$\begin{aligned} & \text{maximize} && t \\ & \text{subject to} && t \leq \det(P)^{1/n}, \\ & && \|Px_i - c\|_2 \leq 1, \quad i = 1, \dots, m, \\ & && P \succeq 0, \end{aligned} \tag{11.23}$$

and then directly into *Fusion* code:

Listing 11.12: *Fusion* implementation of model (11.23).

```
std::pair<std::shared_ptr<ndarray<double, 1>>, std::shared_ptr<ndarray<double, 1>>>
  lowerjohn_outer(std::shared_ptr<ndarray<double, 2>> x)
{
  Model::t M = new Model("lowerjohn_outer");
  int m = x->size(0);
  int n = x->size(1);

  // Setup variables
  Variable::t t = M->variable("t", 1, Domain::greaterThan(0.0));
  Variable::t P = det_rootn(M, t, n);
  Variable::t c = M->variable("c", n, Domain::unbounded());

  // (1, Px-c) \in Q
  M->constraint(Expr::hstack(
    Expr::ones(m), Expr::sub(Expr::mul(x, P),
    Var::reshape(Var::repeat(c, m), new_array_ptr<int, 1>({m, n}))),
    ↵),
    ↵,
    Domain::inQCone());
```

(continues on next page)

```

// Objective: Maximize t
M->objective(ObjectiveSense::Maximize, t);
M->solve();

return std::make_pair(P->level(), c->level());
}

```

11.6.3 Bound on the Determinant Root

It remains to show how to express the bounds on $\det(X)^{1/n}$ for a symmetric positive definite $n \times n$ matrix X using PSD and conic quadratic variables. We want to model the set

$$C = \{(X, t) \in \mathcal{S}_+^n \times \mathbb{R} \mid t \leq \det(X)^{1/n}\}. \quad (11.24)$$

A standard approach when working with the determinant of a PSD matrix is to consider a semidefinite cone

$$\begin{pmatrix} X & Z \\ Z^T & \text{Diag}(Z) \end{pmatrix} \succeq 0 \quad (11.25)$$

where Z is a matrix of additional variables and where we intuitively identify $\text{Diag}(Z) = \{\lambda_1, \dots, \lambda_n\}$ with the eigenvalues of X . With this in mind, we are left with expressing the constraint

$$t \leq (\lambda_1 \cdots \lambda_n)^{1/n}. \quad (11.26)$$

This is easy to implement recursively using rotated quadratic cones when n is a power of 2. In general it is convenient to express (11.26) as a composition of power cones, see [BenTalN01] or [Modeling Cookbook](#).

Listing 11.13: Approaching the determinant, see (11.25).

```

Variable::t det_rootn(Model::t M, Variable::t t, int n)
{
    // Setup variables
    Variable::t Y = M->variable(Domain::inPSDCone(2 * n));

    Variable::t X = Y->slice(new_array_ptr<int,1>({0, 0}), new_array_ptr<int,1>({n, n}
↪));
    Variable::t Z = Y->slice(new_array_ptr<int,1>({0, n}), new_array_ptr<int,1>({n, 2
↪ * n}));
    Variable::t DZ = Y->slice(new_array_ptr<int,1>({n, n}), new_array_ptr<int,1>({2 *
↪ n, 2 * n}));

    // Z is lower-triangular
    std::shared_ptr<ndarray<int,2>> low_tri( new ndarray<int,2>( shape_t<2>(n*(n-1)/2,
↪ 2) ));
    int k = 0;
    for(int i = 0; i < n; i++)
        for(int j = i+1; j < n; j++)
            (*low_tri)(k,0) = i, (*low_tri)(k,1) = j, ++k;
    M->constraint(Z->pick(low_tri), Domain::equalsTo(0.0));
    // DZ = Diag(Z)
    M->constraint(Expr::sub(DZ, Expr::mulElm(Z, Matrix::eye(n))), Domain::equalsTo(0.
↪));

    // t^n <= (Z11*Z22*...*Znn)
    geometric_mean(M, DZ->diag(), t);
}

```

(continues on next page)

```

// Return an n x n PSD variable which satisfies t <= det(X)^(1/n)
return X;
}

```

Listing 11.14: Bounding the geometric mean, see (11.26).

```

void geometric_mean(Model::t M, Variable::t x, Variable::t t)
{
    int n = (int) x->getSize();
    if (n==1)
        M->constraint(Expr::sub(t, x), Domain::lessThan(0.0));
    else
    {
        Variable::t t2 = M->variable();
        M->constraint(Var::hstack(t2, x->index(n-1), t), Domain::inPPowerCone(1-1.0/n));
        geometric_mean(M, x->slice(0,n-1), t2);
    }
}

```

11.7 SUDOKU

SUDOKU is a famous simple yet mind-blowing game. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called *boxes*, *blocks*, *regions*, or *sub-squares*) contains all of the digits from 1 to 9. For more information see <http://en.wikipedia.org/wiki/Sudoku>. Here is a simple example:

				4				
	5	8			3			
	1		2	8		9		
	7	3	1			8	4	
	4	1			9	2	7	
		4		6	5		8	
			4			1	6	
				9				

A simple unsolved Sudoku

3	2	9	5	4	7	6	1	8
6	5	8	9	1	3	4	2	7
4	1	7	2	8	6	9	5	3
9	7	3	1	5	2	8	4	6
5	6	2	8	7	4	3	9	1
8	4	1	6	3	9	2	7	5
1	9	4	3	6	5	7	8	2
7	3	5	4	2	8	1	6	9
2	8	6	7	9	1	5	3	4

The solution

In a more general setting we are given a grid of dimension $n \times n$, with $n = m^2, m \in \mathbb{N}$. Each cell (i, j) must be filled with an integer $y_{ij} \in [1, n]$. Along each row and each column there must be no repetitions. No repetitions are allowed also in each sub-grid with corners $\{(mt, ml), (m(t+1)-1, m(l+1)-1)\}$, for $t, l = 0, \dots, m-1$ (we index cells from $(0, 0)$).

In general, each SUDOKU instance comes with a set F of predetermined values which:

- reduce the complexity of the game by removing symmetries and guiding the initial moves of the player;
- ensure that there will be a unique solution.

We represent the set F as list of triplets (i, j, v) , meaning that the cell (i, j) contains the value v .

Note that SUDOKU is a **feasibility** problem. A typical Integer Programming formulation is straightforward: let x_{ijk} be a binary variable that takes value 1 if k is written in cell (i, j) . Then we look for a feasible solution of a system of constraints given below.

SUDOKU is a typical assignment problem. Its constraints are commonly found in optimization problems concerning scheduling or resource allocation. SUDOKU has also been a nice problem to fiddle with for many researchers in the optimization community. Indeed, its simple structure and the easy way in which the results can be tested make it a perfect test problem.

We will approach SUDOKU as a standard integer linear program, and we will show how easily and elegantly it can be implemented in *Fusion*.

Mathematical Formulation

In this section we formulate SUDOKU as a mixed-integer linear optimization problem. Let's introduce a binary variable x_{ijk} that takes value 1 if k is written in the cell (i, j) , or 0 otherwise. We first ask that for each cell exactly one digit is selected:

$$\sum_{k=0}^{n-1} x_{ijk} = 1, \quad i, j = 0, \dots, n-1. \quad (11.27)$$

Similar constraints can be used to force each digit to appear only once in each row or column:

$$\begin{aligned} \sum_{i=0}^{n-1} x_{ijk} &= 1, & j, k &= 0, \dots, n-1, \\ \sum_{j=0}^{n-1} x_{ijk} &= 1, & i, k &= 0, \dots, n-1. \end{aligned} \quad (11.28)$$

To force a digit to appear only once in each sub-grid we can use the following

$$\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1 \quad k = 0, \dots, n-1 \text{ and } t, l = 0, \dots, m-1 \quad (11.29)$$

If a cell (i, j) has a predetermined value, i.e. $(i, j, k) \in F$ then we set

$$x_{ijk} = 1.$$

Summarizing, and considering that there is no objective function to minimize, the optimization model for the SUDOKU problem takes the form

$$\begin{aligned} &\min 0 \\ &\text{s.t.} \\ &\sum_{i=0}^{n-1} x_{ijk} = 1, & j, k &= 0, \dots, n-1, \\ &\sum_{j=0}^{n-1} x_{ijk} = 1, & i, k &= 0, \dots, n-1, \\ &\sum_{k=0}^{n-1} x_{ijk} = 1, & i, j &= 0, \dots, n-1, \\ &\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_{(i+tm)(j+tl)k} = 1, & k &= 0, \dots, n-1 \text{ and } \\ & & & t, l = 0, \dots, m-1, \\ &x_{ijk} = 1, & \forall (i, j, k) \in F. \end{aligned} \quad (11.30)$$

Implementation with *Fusion*

The implementation in *Fusion* is straightforward. First, we represent the variable x using a three dimensional *Fusion* variable:

```
Variable::t X = M->variable("X", new_array_ptr<int>, 1>({n, n, n}), Domain::
  ↪binary());
```

Then we can define constraints (11.27) and (11.28) simply using the *Expr.sum* operator, that allows to sum the elements of an expression (in this case of the variable itself) along arbitrary dimensions. The code reads:

```
//each value only once per dimension
for (int d = 0; d < m; d++)
  M->constraint( Expr::sum(X, d), Domain::equalsTo(1.) );
```

The last set of constraints (11.29), i.e. the sum over block, needs a little more effort: we must loop over all blocks and select the proper slice:

```

//each number must appear only once in a block
for (int k = 0; k < n ; k++)
    for (int i = 0; i < m ; i++)
        for (int j = 0; j < m ; j++)
            M->constraint( Expr::sum( X->slice( new_array_ptr<int, 1>({i * m, j * m, k}),
                                                new_array_ptr<int, 1>({(i + 1)*m, (j +
↪1)*m, k + 1}) ) ),
                        Domain::equalsTo(1.) );

```

To set the triplets given in the set F we can use the `Variable.pick` method that returns a one dimensional view of an arbitrary set of elements of the variable.

```

auto fixed = std::shared_ptr< ndarray<int, 2> >( new ndarray<int, 2>( shape(nfixed,
↪3) ) );

for (int i = 0; i < nfixed; i++)
    for (int d = 0; d < m; d++)
        (*fixed)(i, d) = (*hr_fixed)(i, d) - 1;

M->constraint( X->pick( fixed ) , Domain::equalsTo(1.0) ) ;

```

SUDOKU: the complete example code.

The complete code for the SUDOKU problem is shown in [Listing 11.15](#).

Listing 11.15: *Fusion* implementation to solve SUDOKU.

```

#include <iostream>
#include <sstream>
#include <cmath>

#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;

void print_solution(int n, Variable::t X)
{
    using namespace std;

    cout << "\n";
    int m( std::sqrt(n) );
    for (int i = 0; i < n; i++)
    {
        stringstream ss;

        for (int j = 0; j < n; j++)
        {
            if (j % m == 0) ss << " |";

            for (int k = 0; k < n; k++)
            {
                auto x = X->index(new_array_ptr<int, 1>({i, j, k}))->level();
                if ( (*x)[0] > 0.5 )
                {
                    ss << " " << (k + 1);
                    break;
                }
            }
        }
    }
}

```

(continues on next page)

```

    }
}
cout << ss.str() << " |";

cout << "\n";
if ((i + 1) % m == 0)
    cout << "\n";
}
}

int main(int argc, char ** argv)
{

    int m = 3;
    int n = m * m;

    //fixed cells in human readable (i.e. 1-based) format
    auto hr_fixed = new_array_ptr<int, 2>(
    { {1, 5, 4},
      {2, 2, 5}, {2, 3, 8}, {2, 6, 3},
      {3, 2, 1}, {3, 4, 2}, {3, 5, 8}, {3, 7, 9},
      {4, 2, 7}, {4, 3, 3}, {4, 4, 1}, {4, 7, 8}, {4, 8, 4},
      {6, 2, 4}, {6, 3, 1}, {6, 6, 9}, {6, 7, 2}, {6, 8, 7},
      {7, 3, 4}, {7, 5, 6}, {7, 6, 5}, {7, 8, 8},
      {8, 4, 4}, {8, 7, 1}, {8, 8, 6},
      {9, 5, 9}
    }
    );

    int nfixed = hr_fixed->size() / m;

    Model::t M = new Model("SUDOKU"); auto _M = finally([&]() { M->dispose(); });

    M->setLogHandler([ = ](const std::string & msg) { std::cout << msg << std::flush; }
    ↪);

    Variable::t X = M->variable("X", new_array_ptr<int, 1>({n, n, n}), Domain::
    ↪binary());

    //each value only once per dimension
    for (int d = 0; d < m; d++)
        M->constraint( Expr::sum(X, d), Domain::equalsTo(1.) );

    //each number must appear only once in a block
    for (int k = 0; k < n ; k++)
        for (int i = 0; i < m ; i++)
            for (int j = 0; j < m ; j++)
                M->constraint( Expr::sum( X->slice( new_array_ptr<int, 1>({i * m, j * m, k}),
                ↪new_array_ptr<int, 1>({(i + 1)*m, (j +
                ↪1)*m, k + 1}) ) ),
                Domain::equalsTo(1.) );

    auto fixed = std::shared_ptr< ndarray<int, 2> >( new ndarray<int, 2>( shape(nfixed,
    ↪3) ) );

    for (int i = 0; i < nfixed; i++)

```

```

    for (int d = 0; d < m; d++)
        (*fixed)(i, d) = (*hr_fixed)(i, d) - 1;

M->constraint( X->pick( fixed ) , Domain::equalsTo(1.0) ) ;

M->solve();

//print the solution, if any...
if ( M->getPrimalSolutionStatus() == SolutionStatus::Optimal )
    print_solution(n, X);
else
    std::cout << "No solution found!\n";

return 0;
}

```

The problem instance corresponding to Fig. ?? is hard-coded for the sake of simplicity. It will produce the following output

```

Problem
  Name           : SUDOKU
  Objective sense : min
  Type           : LO (linear optimization problem)
  Constraints     : 350
  Cones          : 0
  Scalar variables : 1000
  Matrix variables : 0
  Integer variables : 729

Optimizer started.
Mixed integer optimizer started.
Threads used: 2
Presolve started.
Presolve terminated. Time = 0.00
Presolved problem: 0 variables, 0 constraints, 0 non-zeros
Presolved problem: 0 general integer, 0 binary, 0 continuous
Clique table size: 0
BRANCHES RELAXS  ACT_NDS  DEPTH    BEST_INT_OBJ          BEST_RELAX_OBJ          REL_GAP(
->%)  TIME
0      1      0      0      0.0000000000e+00      0.0000000000e+00      0.
->00e+00  0.0
An optimal solution satisfying the relative gap tolerance of 1.00e-02(%) has been
->located.
The relative gap is 0.00e+00(%).
An optimal solution satisfying the absolute gap tolerance of 0.00e+00 has been
->located.
The absolute gap is 0.00e+00.

Objective of best integer solution : 0.000000000000e+00
Best objective bound                : -0.000000000000e+00
Construct solution objective        : Not employed
Construct solution # roundings      : 0
User objective cut value           : 0
Number of cuts generated            : 0
Number of branches                  : 0
Number of relaxations solved        : 1
Number of interior point iterations: 0

```

(continues on next page)

```

Number of simplex iterations      : 0
Time spend presolving the root    : 0.00
Time spend in the heuristic       : 0.00
Time spend in the sub optimizers  : 0.00
    Time spend optimizing the root : 0.00
Mixed integer optimizer terminated. Time: 0.02

```

```

Optimizer terminated. Time: 0.02

```

```

| 3 2 9 | 5 4 7 | 6 1 8 |
| 6 5 8 | 9 1 3 | 4 2 7 |
| 4 1 7 | 2 8 6 | 9 5 3 |

```

```

| 9 7 3 | 1 5 2 | 8 4 6 |
| 5 6 2 | 8 7 4 | 3 9 1 |
| 8 4 1 | 6 3 9 | 2 7 5 |

```

```

| 1 9 4 | 3 6 5 | 7 8 2 |
| 7 3 5 | 4 2 8 | 1 6 9 |
| 2 8 6 | 7 9 1 | 5 3 4 |

```

11.8 Travelling Salesman Problem (TSP)

The *Travelling Salesman Problem* is one of the most famous and studied problems in combinatorics and integer optimization. In this case study we shall:

- show how to compactly define a model with *Fusion*;
- implement an iterative algorithm that solves a sequence of optimization problems;
- modify an optimization problem by adding more constraints;
- show how to access the solution of an optimization problem.

The material presented in this section draws inspiration from [Pat03].

In a TSP instance we are given a directed graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs. To each arc $(i, j) \in A$ corresponds a nonnegative cost c_{ij} . The goal is to find a minimum cost *Hamilton cycle* in G , that is a closed tour passing through each node exactly once. For example, consider the small directed graph in Fig. 11.8.

Its corresponding adjacency and cost matrices A and c are:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad c = \begin{bmatrix} 0 & 1 & 0.1 & 0.1 \\ 0.1 & 0 & 1 & 0 \\ 0 & 0.1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Typically, the problem is modeled introducing a set of binary variables x_{ij} such that

$$x_{ij} = \begin{cases} 0 & \text{if arc } (i, j) \text{ is in the tour,} \\ 1 & \text{otherwise.} \end{cases}$$

Now we can introduce the following simple model:

$$\begin{aligned} \min \quad & \sum_{i,j} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_i x_{ij} = 1 \quad \forall j = 1, \dots, n, \\ & \sum_j x_{ij} = 1 \quad \forall i = 1, \dots, n, \\ & x_{ij} \leq A_{ij} \quad \forall i, j, \\ & x_{ij} \in \{0, 1\} \quad \forall i, j. \end{aligned} \tag{11.31}$$

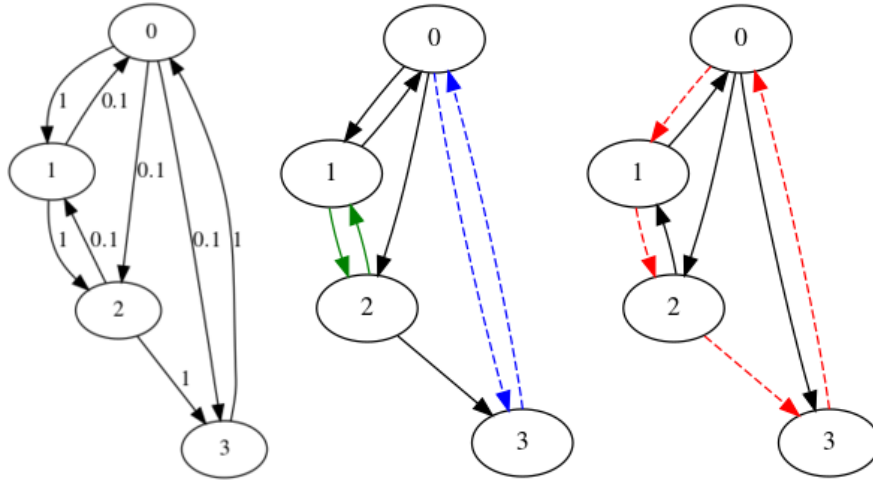


Fig. 11.8: (Left) a directed graph with costs. (Middle) The minimum cycle cover found in the first iteration. (Right) The minimum cost travelling salesman tour.

It describes the constraint that every vertex has exactly one incoming and one outgoing arc in the tour, and that only arcs present in the graph can be chosen. Problem (11.31) can be easily implemented in *Fusion*:

```
Model::t M = new Model();
auto M_ = finally([&]() { M->dispose(); });

auto x = M->variable("x", Set::make(n,n), Domain::binary());

M->constraint(Expr::sum(x, 0), Domain::equalsTo(1.0));
M->constraint(Expr::sum(x, 1), Domain::equalsTo(1.0));
M->constraint(x, Domain::lessThan( A ));

M->objective(ObjectiveSense::Minimize, Expr::dot(C, x));
```

Note in particular how:

- we can sum over rows and/or columns using the *Expr.sum* function;
- we use *Expr.dot* to compute the objective function.

The solution to problem (11.31) is not necessarily a closed tour. In fact (11.31) models another problem known as *minimum cost cycle cover*, whose solution may consist of more than one cycle. In our example we get the solution depicted in Fig. 11.8, i.e. there are two loops, namely $0 \rightarrow 3 \rightarrow 0$ and $1 \rightarrow 2 \rightarrow 1$.

A solution to (11.31) solves the TSP problem if and only if it consists of a single cycle. One classical approach ensuring this is the so-called *subtour elimination*: once we found a solution of (11.31) composed of at least two cycles, we add constraints that explicitly avoid that particular solution:

$$\sum_{(i,j) \in c} x_{ij} \leq |c| - 1 \quad \forall c \in C. \quad (11.32)$$

Thus the problem we want to solve at each step is

$$\begin{aligned} \min \quad & \sum_{i,j} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_i x_{ij} = 1 & \forall j = 1, \dots, n, \\ & \sum_j x_{ij} = 1 & \forall i = 1, \dots, n, \\ & x_{ij} \leq A_{ij} & \forall i, j, \\ & x_{ij} \in \{0, 1\} & \forall i, j, \\ & \sum_{(i,j) \in c} x_{ij} \leq |c| - 1 & \forall c \in C, \end{aligned} \quad (11.33)$$

where C is the set of cycles in all the cycle covers we have seen so far. The overall solution scheme is the following:

1. set C as the empty set,
2. solve problem (11.33),
3. **if** x has only one cycle **stop**,
4. **else** add the cycles of x to C and **goto** 2.

Cycle detection is a fairly easy task and we omit the procedure here for the sake of simplicity. Now we show how to add a constraint for each cycle. Since we have the list of arcs, and each one corresponds to a variable x_{ij} , we can use the function *Variable.pick* to compactly define constraints of the form (11.32):

```
for (auto c : cycles)
{
    int csize = c.size();

    auto tmp = std::shared_ptr<monty::ndarray<int, 2> >(new ndarray<int, 2>(
↳shape(csize, 2)) );
    for (auto i = 0; i < csize; ++i)
    {
        (*tmp)(i, 0) = std::get<0>(c[i]);
        (*tmp)(i, 1) = std::get<1>(c[i]);
    }

    M->constraint(Expr::sum(x->pick(tmp)), Domain::lessThan( 1.0 * csize - 1 ));
}
```

Executing our procedure will yield the following output:

```
it #1 - solution cost: 2.200000

cycles:
[0,3] - [3,0] -
[1,2] - [2,1] -

it #2 - solution cost: 4.000000

cycles:
[0,1] - [1,2] - [2,3] - [3,0] -

solution:
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
```

Thus we first discover the two-cycle solution; then the second iteration is forced not to include those cycles, and a new solution is located. This time it consists of one loop, and as expected the cost is higher. The solution is depicted in Fig. 11.8.

Formulation (11.33) can be improved in some cases by exploiting the graph structure. Some simple tricks follow.

Self-loops

Self-loops are never part of a TSP tour. Typically self-loops are removed by penalizing them with a huge cost c_{ii} . Although this works in practice, it is more advisable to just fix the corresponding variables to zero, i.e.

$$x_{ii} = 0 \quad \forall i = 1, \dots, n. \quad (11.34)$$

This removes redundant variables, and avoids unnecessarily large coefficients that can negatively affect the solver.

Constraints (11.34) are easily implemented as follows:

```
M->constraint(x->diag(), Domain::equalsTo(0.));
```

Two-arc loops removal

In networks with more than two nodes two-loop arcs can also be ignored. They are simple to detect and their number is of the same order as the size of the graph. The constraints we need to add are:

$$x_{ij} + x_{ji} \leq 1 \quad \forall i, j = 1, \dots, n. \quad (11.35)$$

Constraints (11.35) are easily implemented as follows:

```
M->constraint(Expr::add(x, x->transpose()), Domain::lessThan(1.0));
```

The complete working example

Listing 11.16: The complete code for the TSP examples.

```
void tsp(int n, Matrix::t A, Matrix::t C, bool remove_1_hop_loops, bool remove_2_hop_
→loops)
{
    Model::t M = new Model();
    auto M_ = finally([&]() { M->dispose(); });

    auto x = M->variable("x", Set::make(n,n), Domain::binary());

    M->constraint(Expr::sum(x, 0), Domain::equalsTo(1.0));
    M->constraint(Expr::sum(x, 1), Domain::equalsTo(1.0));
    M->constraint(x, Domain::lessThan( A ));

    M->objective(ObjectiveSense::Minimize, Expr::dot(C, x));

    if (remove_1_hop_loops)
        M->constraint(x->diag(), Domain::equalsTo(0.));

    if (remove_2_hop_loops)
        M->constraint(Expr::add(x, x->transpose()), Domain::lessThan(1.0));

    int it = 0;
    while (true)
    {
        M->solve();
        it++;

        typedef std::vector< std::tuple<int, int> > cycle_t;
        std::list< cycle_t > cycles;
```

(continues on next page)

```

auto xlevel = x->level();

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        if ( (*xlevel)[i * n + j] <= 0.5 )
            continue;

        bool found = false;
        for (auto && c : cycles)
        {
            for (auto && cc : c)
            {
                if ( i == std::get<0>(cc) || i == std::get<1>(cc) ||
                    j == std::get<0>(cc) || j == std::get<1>(cc) )
                {
                    c.push_back( std::make_tuple(i, j) );
                    found = true;
                    break;
                }
            }
            if (found) break;
        }

        if (!found)
            cycles.push_back(cycle_t(1, std::make_tuple(i, j)));
    }

std::cout << "Iteration " << it << "\n";
for (auto c : cycles) {
    for (auto cc : c)
        std::cout << "(" << std::get<0>(cc) << "," << std::get<1>(cc) << ") ";
    std::cout << "\n";
}

if (cycles.size() == 1) break;

for (auto c : cycles)
{
    int csize = c.size();

    auto tmp = std::shared_ptr<monty::ndarray<int, 2> >(new ndarray<int, 2>(
↳shape(csize, 2)) );
    for (auto i = 0; i < csize; ++i)
    {
        (*tmp)(i, 0) = std::get<0>(c[i]);
        (*tmp)(i, 1) = std::get<1>(c[i]);
    }

    M->constraint(Expr::sum(x->pick(tmp)), Domain::lessThan( 1.0 * csize - 1 ));
}
}
try {
    auto xlevel = x->level();
    std::cout << "Solution\n";

```

```

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            std::cout << (int) (*xlevel)(i * n + j);
        std::cout << "\n";
    }
} catch (...) {}
}

int main()
{
    auto A_i = new_array_ptr<int, 1>({0, 1, 2, 3, 1, 0, 2, 0});
    auto A_j = new_array_ptr<int, 1>({1, 2, 3, 0, 0, 2, 1, 3});

    auto C_v = new_array_ptr<double, 1>({1., 1., 1., 1., 0.1, 0.1, 0.1, 0.1});

    int n = 4;
    tsp(n, Matrix::sparse(n, n, A_i, A_j, 1.), Matrix::sparse(n, n, A_i, A_j, C_v),
    →true, false);
    tsp(n, Matrix::sparse(n, n, A_i, A_j, 1.), Matrix::sparse(n, n, A_i, A_j, C_v),
    →true, true);

    return 0;
}

```

11.9 Nearest Correlation Matrix Problem

A *correlation matrix* is a symmetric positive definite matrix with unit diagonal. This term has origins in statistics, since the matrix whose entries are the correlation coefficients of a sequence of random variables has all these properties.

In this section we study variants of the problem of approximating a given symmetric matrix A with correlation matrices:

- find the correlation matrix X nearest to A in the *Frobenius norm*,
- find an approximation of the form $D + X$ where D is a diagonal matrix with positive diagonal and X is a positive semidefinite matrix of low rank, using the combination of Frobenius and *nuclear norm*.

Both problems are related to *portfolio optimization*, where one can often have a matrix A that only approximates the correlations of stocks. For subsequent optimizations one would like to approximate A with a correlation matrix or, in the factor model, with $D + VV^T$ with VV^T of small rank.

11.9.1 Nearest correlation with the Frobenius norm

The Frobenius norm of a real matrix M is defined as

$$\|M\|_F = \left(\sum_{i,j} M_{i,j}^2 \right)^{1/2}$$

and with respect to this norm our optimization problem can be expressed simply as:

$$\begin{aligned} & \text{minimize} && \|A - X\|_F \\ & \text{subject to} && \mathbf{diag}(X) = e, \\ & && X \succeq 0. \end{aligned} \tag{11.36}$$

We can exploit the symmetry of A and X to get a compact vector representation. To this end we make use of the following mapping from a symmetric matrix to a flattened vector containing the (scaled) lower

triangular part of the matrix:

$$\begin{aligned} \text{vec} : & \quad \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{n(n+1)/2} \\ \text{vec}(M) = & \quad (\alpha_{11}M_{11}, \alpha_{21}M_{21}, \alpha_{22}M_{22}, \dots, \alpha_{n1}M_{n1}, \dots, \alpha_{nn}M_{nn}) \\ \alpha_{ij} = & \quad \begin{cases} 1 & j = i \\ \sqrt{2} & j < i \end{cases} \end{aligned} \quad (11.37)$$

Note that $\|M\|_F = \|\text{vec}(M)\|_2$. The *Fusion* implementation of `vec` is as follows:

Listing 11.17: Implementation of function `vec` in (11.37).

```
Expression::t vec(Expression::t e)
{
    int N = (*e->getShape())[0];
    int dim = N * (N + 1) / 2;

    auto msubi = new_array_ptr<int, 1>(dim);
    auto msubj = new_array_ptr<int, 1>(dim);
    auto mcof = new_array_ptr<double, 1>(dim);

    for (int i = 0, k = 0; i < N; ++i)
        for (int j = 0; j < i + 1; ++j, ++k)
        {
            (*msubi)[k] = k;
            (*msubj)[k] = i * N + j;
            (*mcof)[k] = (i == j) ? 1.0 : std::sqrt(2.0);
        }

    Matrix::t S = Matrix::sparse(N * (N + 1) / 2, N * N, msubi, msubj, mcof);
    return Expr::mul(S, Expr::reshape(e, N * N));
}
```

That leads to an optimization problem with both conic quadratic and semidefinite constraints:

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && (t, \text{vec}(A - X)) \in \mathcal{Q}, \\ & && \text{diag}(X) = e, \\ & && X \succeq 0. \end{aligned} \quad (11.38)$$

Code example

Listing 11.18: Implementation of problem (11.38).

```
void nearestcorr( std::shared_ptr<ndarray<double, 2>> A)
{
    int N = A->size(0);

    // Create a model
    Model::t M = new Model("NearestCorrelation"); auto _M = finally([&]() { M->
    dispose(); });

    // Setting up the variables
    Variable::t X = M->variable("X", Domain::inPSDCone(N));
    Variable::t t = M->variable("t", 1, Domain::unbounded());

    // (t, vec(A-X)) \in Q
    M->constraint( Expr::vstack(t, vec(Expr::sub(A, X))), Domain::inQCone() );
}
```

(continues on next page)

```

// diag(X) = e
M->constraint(X->diag(), Domain::equalsTo(1.0));

// Objective: Minimize t
M->objective(ObjectiveSense::Minimize, t);

// Solve the problem
M->solve();

// Get the solution values
std::cout << "X = \n"; print_mat(std::cout, X->level());
std::cout << "t = " << *(t->level()->begin()) << std::endl;
}

```

We use the following input

Listing 11.19: Input for the nearest correlation problem.

```

int N = 5;
auto A = new_array_ptr<double, 2>(
{ { 0.0, 0.5, -0.1, -0.2, 0.5},
  { 0.5, 1.25, -0.05, -0.1, 0.25},
  { -0.1, -0.05, 0.51, 0.02, -0.05},
  { -0.2, -0.1, 0.02, 0.54, -0.1},
  { 0.5, 0.25, -0.05, -0.1, 1.25}
});

```

The expected output is the following (small differences may apply):

```

X =
[[ 1.          0.50001941 -0.09999994 -0.20000084  0.50001941]
 [ 0.50001941  1.          -0.04999551 -0.09999154  0.24999101]
 [-0.09999994 -0.04999551  1.          0.01999746 -0.04999551]
 [-0.20000084 -0.09999154  0.01999746  1.          -0.09999154]
 [ 0.50001941  0.24999101 -0.04999551 -0.09999154  1.          ]]

```

11.9.2 Nearest Correlation with Nuclear-norm Penalty

Next, we consider the approximation of A of the form $D + X$ where $D = \mathbf{diag}(w)$, $w \geq 0$ and $X \succeq 0$. We will also aim at minimizing the rank of X . This can be approximated by a relaxed linear objective penalizing the trace $\text{Tr}(X)$ (which in this case is the *nuclear norm* of X and happens to be the sum of its eigenvalues).

The combination of these constraints leads to a problem:

$$\begin{aligned} & \text{minimize} && \|X + \mathbf{diag}(w) - A\|_F + \gamma \text{Tr}(X), \\ & \text{subject to} && X \succeq 0, w \geq 0, \end{aligned}$$

where the parameter γ controls the tradeoff between the quality of approximation and the rank of X .

Exploit the mapping vec defined in (11.37) we can express this problem as:

$$\begin{aligned} & \text{minimize} && t + \gamma \text{Tr}(X) \\ & \text{subject to} && (t, \text{vec}(X + \mathbf{diag}(w) - A)) \in \mathcal{Q}, \\ & && X \succeq 0, w \geq 0. \end{aligned} \tag{11.39}$$

Code example

Listing 11.20: Implementation of problem (11.39).

```
void nearestcorr_nn(
    std::shared_ptr<ndarray<double, 2>> A,
    const std::vector<double> & gammas,
    std::vector<double> & res,
    std::vector<double> & rank)
{
    int N = A->size(0);

    Model::t M = new Model("NucNorm"); auto M_ = monty::finally([&]() { M->dispose(); }
    ↪);

    // Setup variables
    Variable::t t = M->variable("t", 1, Domain::unbounded());
    Variable::t X = M->variable("X", Domain::inPSDCone(N));
    Variable::t w = M->variable("w", N, Domain::greaterThan(0.0));

    // (t, vec (X + diag(w) - A)) in Q
    Expression::t D = Expr::mulElm( Matrix::eye(N), Var::repeat(w, 1, N) );
    M->constraint( Expr::vstack( t, vec(Expr::sub(Expr::add(X, D), A)) ), Domain::
    ↪inQCone() );

    // Trace(X)
    auto TrX = Expr::sum(X->diag());

    for (int k = 0; k < gammas.size(); ++k)
    {
        // Objective: Minimize t + gamma*Tr(X)
        M->objective(ObjectiveSense::Minimize, Expr::add(t, Expr::mul(gammas[k], TrX)));
        M->solve();

        // Find the eigenvalues of X and approximate its rank
        auto d = new_array_ptr<double, 1>(N);
        mosek::LinAlg::sy eig(MSK_UPLO_LO, N, X->level(), d);
        int rn timerank = 0; for (int i = 0; i < N; ++i) if ((*d)[i] > 1e-6) ++rn timerank;

        res[k] = (*(t->level()))[0];
        rank[k] = rn timerank;
    }
}
```

We feed **MOSEK** with the same input as in Sec. 11.9.1. The problem is solved for a range of values γ values, to demonstrate how the penalty term helps achieve a low rank solution. To this extent we report both the rank of X and the residual norm $\|X + \mathbf{diag}(w) - A\|_F$.

```
--- Nearest Correlation with Nuclear Norm---
gamma=0.000000, res=3.076163e-01, rank=4
gamma=0.100000, res=4.251692e-01, rank=2
gamma=0.200000, res=5.112082e-01, rank=1
gamma=0.300000, res=5.298432e-01, rank=1
gamma=0.400000, res=5.592686e-01, rank=1
gamma=0.500000, res=6.045702e-01, rank=1
gamma=0.600000, res=6.764402e-01, rank=1
gamma=0.700000, res=8.009913e-01, rank=1
gamma=0.800000, res=1.062385e+00, rank=1
```

(continues on next page)

```
gamma=0.900000, res=1.129513e+00, rank=0
gamma=1.000000, res=1.129513e+00, rank=0
```

11.10 Semidefinite Relaxation of MIQCQO Problems

In this case study we will discuss a fairly common application for Semidefinite Optimization: to define a continuous semidefinite relaxation of a mixed-integer quadratic optimization problem. This section is based on the method by Park and Boyd [PB15].

We will focus on problems of the form:

$$\begin{aligned} & \text{minimize} && x^T P x + 2q^T x \\ & \text{subject to} && x \in \mathbb{Z}^n \end{aligned} \quad (11.40)$$

where $q \in \mathbb{R}^n$ and $P \in \mathcal{S}_+^{n \times n}$ is positive semidefinite. There are many important problems that can be reformulated as (11.40), for example:

- *integer least squares*: minimize $\|Ax - b\|_2^2$ subject to $x \in \mathbb{Z}^n$,
- *closest vector problem*: minimize $\|v - z\|_2$ subject to $z \in \{Bx \mid x \in \mathbb{Z}^n\}$.

Following [PB15], we can derive a relaxed continuous model. We first relax the integrality constraint

$$\begin{aligned} & \text{minimize} && x^T P x + 2q^T x \\ & \text{subject to} && x_i(x_i - 1) \geq 0 \quad i = 1, \dots, n. \end{aligned}$$

The last constraint is still non-convex. We introduce a new variable $X \in \mathbb{R}^{n \times n}$, such that $X = x \cdot x^T$. This allows us to write an equivalent formulation:

$$\begin{aligned} & \text{minimize} && \text{Tr}(PX) + 2q^T x \\ & \text{subject to} && \mathbf{diag}(X) \geq x, \\ & && X = x \cdot x^T. \end{aligned}$$

To get a conic problem we relax the last constraint and apply the Schur complement. The final relaxation follows:

$$\begin{aligned} & \text{minimize} && \text{Tr}(PX) + 2q^T x \\ & \text{subject to} && \mathbf{diag}(X) \geq x, \\ & && \begin{bmatrix} X & x \\ x^T & 1 \end{bmatrix} \in \mathcal{S}_+^{n+1}. \end{aligned} \quad (11.41)$$

Fusion Implementation

Implementing model (11.41) in *Fusion* is very simple. We assume the input n , P and q . Then we proceed creating the optimization model

```
Model::t M = new Model();
```

The important step is to define a single PSD variable

$$Z = \begin{bmatrix} X & x \\ x^T & 1 \end{bmatrix} \in \mathcal{S}_+^{n+1}.$$

Our code will create Z and two slices that correspond to X and x :

```
Variable::t Z = M->variable("Z", Domain::inPSDCone(n + 1));

Variable::t X = Z->slice(new_array_ptr<int, 1>({0, 0}), new_array_ptr<int, 1>({n, n}
↪));
Variable::t x = Z->slice(new_array_ptr<int, 1>({0, n}), new_array_ptr<int, 1>({n, n_
↪+ 1}));
```

Then we define the constraints:

```
M->constraint( Expr::sub(X->diag(), x), Domain::greaterThan(0.) );
M->constraint( Z->index(n, n), Domain::equalsTo(1.) );
```

The objective function uses several available linear expressions:

```
M->objective( ObjectiveSense::Minimize, Expr::add(
    Expr::sum( Expr::mulElm( P, X ) ),
    Expr::mul( 2.0, Expr::dot(x, q) )
) );
```

Note that the *trace* operator is not directly available in *Fusion*, but it can easily be defined from scratch.

Complete code

Listing 11.21: *Fusion* implementation of model (11.41).

```
Model::t miqcqp_sdo_relaxation(int n, Matrix::t P, const std::shared_ptr<ndarray
↪<double, 1>> & q) {
    Model::t M = new Model();

    Variable::t Z = M->variable("Z", Domain::inPSDCone(n + 1));

    Variable::t X = Z->slice(new_array_ptr<int, 1>({0, 0}), new_array_ptr<int, 1>({n, n}
↪));
    Variable::t x = Z->slice(new_array_ptr<int, 1>({0, n}), new_array_ptr<int, 1>({n, n}
↪+ 1));

    M->constraint( Expr::sub(X->diag(), x), Domain::greaterThan(0.) );
    M->constraint( Z->index(n, n), Domain::equalsTo(1.) );

    M->objective( ObjectiveSense::Minimize, Expr::add(
        Expr::sum( Expr::mulElm( P, X ) ),
        Expr::mul( 2.0, Expr::dot(x, q) )
    ) );

    return M;
}
```

Numerical Examples

We present now some simple numerical experiments for the integer least squares problem:

$$\begin{aligned} &\text{minimize} && \|Ax - b\|_2^2 \\ &\text{subject to} && x \in \mathbb{Z}^n. \end{aligned} \tag{11.42}$$

It corresponds to the problem (11.40) with $P = A^T A$ and $q = -A^T b$. Following [PB15] we will generate the input data by taking all entries of A from the normal distribution $\mathcal{N}(0, 1)$ and setting $b = Ac$ where c comes from the uniform distribution on $[0, 1]$.

We implement the linear algebra operations using the `LinAlg` module available in **MOSEK**.

An integer rounding `xRound` of the solution to (11.41) is a feasible integer solution to (11.42). We can compare it to the actual optimal integer solution `xOpt`, whenever the latter is available. Of course it is very simple to formulate the integer least squares problem in *Fusion*:

```
Model::t int_least_squares(int n, Matrix::t A, const std::shared_ptr<ndarray<double, 1>>
↪ & b) {
    Model::t M = new Model();
```

(continues on next page)

(continued from previous page)

```
Variable::t x = M->variable("x", n, Domain::integral(Domain::unbounded()));
Variable::t t = M->variable("t", 1, Domain::unbounded());

M->constraint( Expr::vstack(t, Expr::sub(Expr::mul(A, x), b)), Domain::inQCone() );
M->objective( ObjectiveSense::Minimize, t );

return M;
}
```

All that remains is to compare the values of the objective function $\|Ax - b\|_2$ for the two solutions.

Listing 11.22: The comparison of two solutions.

```
// problem dimensions
int n = 20;
int m = 2 * n;

auto c = new_array_ptr<double, 1>(n);
auto A = new_array_ptr<double, 1>(n * m);
auto P = new_array_ptr<double, 1>(n * n);
auto b = new_array_ptr<double, 1>(m);
auto q = new_array_ptr<double, 1>(n);

std::generate(A->begin(), A->end(), std::bind(normal_distr, generator));
std::generate(c->begin(), c->end(), std::bind(unif_distr, generator));
std::fill(b->begin(), b->end(), 0.0);
std::fill(q->begin(), q->end(), 0.0);

// P = A^T A
syrk(MSK_UPLO_LO, MSK_TRANSPOSE_YES,
     n, m, 1.0, A, 0., P);
for (int j = 0; j < n; j++) for (int i = j + 1; i < n; i++) (*P)[i * n + j] =
→(*P)[j * n + i];

// q = -P c, b = A c
gemv(MSK_TRANSPOSE_NO, n, n, -1.0, P, c, 0., q);
gemv(MSK_TRANSPOSE_NO, m, n, 1.0, A, c, 0., b);

// Solve the problems
{
    Model::t M = miqcqp_sdo_relaxation(n, Matrix::dense(n, n, P), q);
    Model::t Mint = int_least_squares(n, Matrix::dense(n, m, A)->transpose(), b);
    M->solve();
    Mint->solve();

    auto xRound = M->getVariable("Z")->
        slice(new_array_ptr<int, 1>({0, n}), new_array_ptr<int, 1>({n, n +
→1}))->level();
    for (int i = 0; i < n; i++) (*xRound)[i] = round((*xRound)[i]);
    auto yRound = new_array_ptr<double, 1>(m);
    auto xOpt = Mint->getVariable("x")->level();
    auto yOpt = new_array_ptr<double, 1>(m);
    std::copy(b->begin(), b->end(), yRound->begin());
    std::copy(b->begin(), b->end(), yOpt->begin());
    gemv(MSK_TRANSPOSE_NO, m, n, 1.0, A, xRound, -1.0, yRound); // Ax_round-b
    gemv(MSK_TRANSPOSE_NO, m, n, 1.0, A, xOpt, -1.0, yOpt); // Ax_opt-b
}
```

(continues on next page)

```

    std::cout << M->getSolverDoubleInfo("optimizerTime") << " " << Mint->
↪getSolverDoubleInfo("optimizerTime") << "\n";
    double valRound, valOpt;
    dot(m, yRound, yRound, valRound); dot(m, yOpt, yOpt, valOpt);
    std::cout << sqrt(valRound) << " " << sqrt(valOpt) << "\n";
}

```

Experimentally the objective value for `xRound` approximates the optimal solution with a factor of 1.1-1.4. We refer to [PB15] for a more involved iterative rounding procedure, producing integer solutions of even better quality, and for a detailed discussion of test results.

Chapter 12

Problem Formulation and Solutions

In this chapter we will discuss the following issues:

- The formal, mathematical formulations of the problem types that **MOSEK** can solve and their duals.
- The solution information produced by **MOSEK**.
- The infeasibility certificate produced by **MOSEK** if the problem is infeasible.

For the underlying mathematical concepts, derivations and proofs see the [Modeling Cookbook](#) or any book on convex optimization. This chapter explains how the related data is organized specifically within the **MOSEK** API.

12.1 Linear Optimization

MOSEK accepts linear optimization problems of the form

$$\begin{array}{llllll} \text{minimize} & & c^T x + c^f & & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \end{array} \quad (12.1)$$

where

- m is the number of constraints.
- n is the number of decision variables.
- $x \in \mathbb{R}^n$ is a vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear part of the objective function.
- $c^f \in \mathbb{R}$ is a constant term in the objective
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.

Lower and upper bounds can be infinite, or in other words the corresponding bound may be omitted.

A primal solution (x) is *(primal) feasible* if it satisfies all constraints in (12.1). If (12.1) has at least one primal feasible solution, then (12.1) is said to be (primal) feasible. In case (12.1) does not have a feasible solution, the problem is said to be *(primal) infeasible*.

12.1.1 Duality for Linear Optimization

Corresponding to the primal problem (12.1), there is a dual problem

$$\begin{aligned} & \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ & \text{subject to} && \begin{aligned} A^T y + s_l^x - s_u^x &= c, \\ -y + s_l^c - s_u^c &= 0, \\ s_l^c, s_u^c, s_l^x, s_u^x &\geq 0. \end{aligned} \end{aligned} \quad (12.2)$$

If a bound in the primal problem is plus or minus infinity, the corresponding dual variable is fixed at 0, and we use the convention that the product of the bound value and the corresponding dual variable is 0. This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. In other words:

$$l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0 \text{ and } l_j^x \cdot (s_l^x)_j = 0.$$

A solution

$$(y, s_l^c, s_u^c, s_l^x, s_u^x)$$

to the dual problem is feasible if it satisfies all the constraints in (12.2). If (12.2) has at least one feasible solution, then (12.2) is *(dual) feasible*, otherwise the problem is *(dual) infeasible*.

A solution

$$(x^*, y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$$

is denoted a *primal-dual feasible solution*, if (x^*) is a solution to the primal problem (12.1) and $(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$ is a solution to the corresponding dual problem (12.2). We also define an auxiliary vector

$$(x^c)^* := Ax^*$$

containing the activities of linear constraints.

For a primal-dual feasible solution we define the *duality gap* as the difference between the primal and the dual objective value,

$$\begin{aligned} & c^T x^* + c^f - \{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \} \\ &= \sum_{i=0}^{m-1} [(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*)] \\ &+ \sum_{j=0}^{n-1} [(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*)] \geq 0 \end{aligned} \quad (12.3)$$

where the first relation can be obtained by transposing and multiplying the dual constraints (12.2) by x^* and $(x^c)^*$ respectively, and the second relation comes from the fact that each term in each sum is nonnegative. It follows that the primal objective will always be greater than or equal to the dual objective.

It is well-known that a linear optimization problem has an optimal solution if and only if there exist feasible primal-dual solution so that the duality gap is zero, or, equivalently, that the *complementarity conditions*

$$\begin{aligned} (s_l^c)_i^* ((x_i^c)^* - l_i^c) &= 0, & i = 0, \dots, m-1, \\ (s_u^c)_i^* (u_i^c - (x_i^c)^*) &= 0, & i = 0, \dots, m-1, \\ (s_l^x)_j^* (x_j - l_j^x) &= 0, & j = 0, \dots, n-1, \\ (s_u^x)_j^* (u_j^x - x_j^*) &= 0, & j = 0, \dots, n-1, \end{aligned}$$

are satisfied.

If (12.1) has an optimal solution and **MOSEK** solves the problem successfully, both the primal and dual solution are reported, including a status indicating the exact state of the solution.

12.1.2 Infeasibility for Linear Optimization

Primal Infeasible Problems

If the problem (12.1) is infeasible (has no feasible solution), **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is the certificate of infeasibility, and the primal solution is undefined.

A certificate of primal infeasibility is a feasible solution to the modified dual problem

$$\begin{aligned} & \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\ & \text{subject to} && \\ & && A^T y + s_l^x - s_u^x = 0, \\ & && -y + s_l^c - s_u^c = 0, \\ & && s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \end{aligned} \tag{12.4}$$

such that the objective value is strictly positive, i.e. a solution

$$(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$$

to (12.4) so that

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* > 0.$$

Such a solution implies that (12.4) is unbounded, and that (12.1) is infeasible.

Dual Infeasible Problems

If the problem (12.2) is infeasible (has no feasible solution), **MOSEK** will report a certificate of dual infeasibility: The primal solution reported is the certificate of infeasibility, and the dual solution is undefined.

A certificate of dual infeasibility is a feasible solution to the modified primal problem

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && \hat{l}^c \leq Ax \leq \hat{u}^c, \\ & && \hat{l}^x \leq x \leq \hat{u}^x, \end{aligned} \tag{12.5}$$

where

$$\hat{l}_i^c = \begin{cases} 0 & \text{if } l_i^c > -\infty, \\ -\infty & \text{otherwise,} \end{cases} \quad \text{and} \quad \hat{u}_i^c := \begin{cases} 0 & \text{if } u_i^c < \infty, \\ \infty & \text{otherwise,} \end{cases}$$

and

$$\hat{l}_j^x = \begin{cases} 0 & \text{if } l_j^x > -\infty, \\ -\infty & \text{otherwise,} \end{cases} \quad \text{and} \quad \hat{u}_j^x := \begin{cases} 0 & \text{if } u_j^x < \infty, \\ \infty & \text{otherwise,} \end{cases}$$

such that

$$c^T x < 0.$$

Such a solution implies that (12.5) is unbounded, and that (12.2) is infeasible.

In case that both the primal problem (12.1) and the dual problem (12.2) are infeasible, **MOSEK** will report only one of the two possible certificates — which one is not defined (**MOSEK** returns the first certificate found).

12.1.3 Minimalization vs. Maximalization

When the objective sense of problem (12.1) is maximization, i.e.

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

the objective sense of the dual problem changes to minimization, and the domain of all dual variables changes sign in comparison to (12.2). The dual problem thus takes the form

$$\begin{aligned} & \text{minimize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ & \text{subject to} && A^T y + s_l^x - s_u^x = c, \\ & && -y + s_l^c - s_u^c = 0, \\ & && s_l^c, s_u^c, s_l^x, s_u^x \leq 0. \end{aligned}$$

This means that the duality gap, defined in (12.3) as the primal minus the dual objective value, becomes nonpositive. It follows that the dual objective will always be greater than or equal to the primal objective. The primal infeasibility certificate will be reported by **MOSEK** as a solution to the system

$$\begin{aligned} A^T y + s_l^x - s_u^x &= 0, \\ -y + s_l^c - s_u^c &= 0, \\ s_l^c, s_u^c, s_l^x, s_u^x &\leq 0, \end{aligned} \tag{12.6}$$

such that the objective value is strictly negative

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* < 0.$$

Similarly, the certificate of dual infeasibility is an x satisfying the requirements of (12.5) such that $c^T x > 0$.

12.2 Conic Optimization

Conic optimization is an extension of linear optimization (see Sec. 12.1) allowing conic domains to be specified for subsets of the problem variables. A conic optimization problem to be solved by **MOSEK** can be written as

$$\begin{aligned} & \text{minimize} && c^T x + c^f \\ & \text{subject to} && \begin{array}{lll} l^c & \leq & Ax & \leq & u^c, \\ l^x & \leq & x & \leq & u^x, \\ & & x \in \mathcal{K}, \end{array} \end{aligned} \tag{12.7}$$

where

- m is the number of constraints.
- n is the number of decision variables.
- $x \in \mathbb{R}^n$ is a vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear part of the objective function.
- $c^f \in \mathbb{R}$ is a constant term in the objective
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.

Lower and upper bounds can be infinite, or in other words the corresponding bound may be omitted.

The set \mathcal{K} is a Cartesian product of convex cones, namely $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_p$. Having the domain restriction $x \in \mathcal{K}$, is thus equivalent to

$$x^t \in \mathcal{K}_t \subseteq \mathbb{R}^{n_t},$$

where $x = (x^1, \dots, x^p)$ is a partition of the problem variables. Please note that the n -dimensional Euclidean space \mathbb{R}^n is a cone itself, so simple linear variables are still allowed. The user only needs to specify subsets of variables which belong to non-trivial cones.

In this section we discuss the formulations which apply to the following cones supported by **MOSEK**:

- The set \mathbb{R}^n .
- The zero cone $\{(0, \dots, 0)\}$.
- Quadratic cone

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n : x_1 \geq \sqrt{\sum_{j=2}^n x_j^2} \right\}.$$

- Rotated quadratic cone

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n : 2x_1x_2 \geq \sum_{j=3}^n x_j^2, \quad x_1 \geq 0, \quad x_2 \geq 0 \right\}.$$

- Primal exponential cone

$$K_{\text{exp}} = \{x \in \mathbb{R}^3 : x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0\}$$

as well as its dual

$$K_{\text{exp}}^* = \{x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0\}.$$

- Primal power cone (with parameter $0 < \alpha < 1$)

$$\mathcal{P}_n^{\alpha, 1-\alpha} = \left\{ x \in \mathbb{R}^n : x_1^\alpha x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^n x_j^2}, \quad x_1, x_2 \geq 0 \right\}$$

as well as its dual

$$(\mathcal{P}_n^{\alpha, 1-\alpha})^* = \left\{ x \in \mathbb{R}^n : \left(\frac{x_1}{\alpha}\right)^\alpha \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^n x_j^2}, \quad x_1, x_2 \geq 0 \right\}.$$

MOSEK supports also the cone of positive semidefinite matrices. Since that is handled through a separate interface, we discuss it in [Sec. 12.3](#).

12.2.1 Duality for Conic Optimization

Corresponding to the primal problem (12.7), there is a dual problem

$$\begin{aligned} & \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ & \text{subject to} && \\ & && A^T y + s_l^x - s_u^x + s_n^x = c \\ & && -y + s_l^c - s_u^c = 0, \\ & && s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ & && s_n^x \in \mathcal{K}^*, \end{aligned} \tag{12.8}$$

where the dual cone \mathcal{K}^* is a Cartesian product of the cones dual to \mathcal{K}_t . In practice this means that s_n^x has one entry for each entry in x . Please note that the dual problem of the dual problem is identical to the original primal problem.

If a bound in the primal problem is plus or minus infinity, the corresponding dual variable is fixed at 0, and we use the convention that the product of the bound value and the corresponding dual variable is 0. This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. In other words:

$$l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0 \text{ and } l_j^x \cdot (s_l^x)_j = 0.$$

A solution

$$(y, s_l^c, s_u^c, s_l^x, s_u^x, s_n^x)$$

to the dual problem is feasible if it satisfies all the constraints in (12.8). If (12.8) has at least one feasible solution, then (12.8) is *(dual) feasible*, otherwise the problem is *(dual) infeasible*.

A solution

$$(x^*, y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$$

is denoted a *primal-dual feasible solution*, if (x^*) is a solution to the primal problem (12.7) and $(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$ is a solution to the corresponding dual problem (12.8). We also define an auxiliary vector

$$(x^c)^* := Ax^*$$

containing the activities of linear constraints.

For a primal-dual feasible solution we define the *duality gap* as the difference between the primal and the dual objective value,

$$\begin{aligned} & c^T x^* + c^f - \{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* + c^f \} \\ &= \sum_{i=0}^{m-1} [(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*)] \\ &+ \sum_{j=0}^{n-1} [(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*)] + \sum_{j=0}^{n-1} (s_n^x)_j^* x_j^* \geq 0 \end{aligned} \quad (12.9)$$

where the first relation can be obtained by transposing and multiplying the dual constraints (12.2) by x^* and $(x^c)^*$ respectively, and the second relation comes from the fact that each term in each sum is nonnegative. It follows that the primal objective will always be greater than or equal to the dual objective.

It is well-known that, under some non-degeneracy assumptions that exclude ill-posed cases, a conic optimization problem has an optimal solution if and only if there exist feasible primal-dual solution so that the duality gap is zero, or, equivalently, that the *complementarity conditions*

$$\begin{aligned} (s_l^c)_i^* ((x_i^c)^* - l_i^c) &= 0, & i = 0, \dots, m-1, \\ (s_u^c)_i^* (u_i^c - (x_i^c)^*) &= 0, & i = 0, \dots, m-1, \\ (s_l^x)_j^* (x_j - l_j^x) &= 0, & j = 0, \dots, n-1, \\ (s_u^x)_j^* (u_j^x - x_j^*) &= 0, & j = 0, \dots, n-1, \\ \sum_{j=0}^{n-1} (s_n^x)_j^* x_j^* &= 0. \end{aligned} \quad (12.10)$$

are satisfied.

If (12.7) has an optimal solution and **MOSEK** solves the problem successfully, both the primal and dual solution are reported, including a status indicating the exact state of the solution.

12.2.2 Infeasibility for Conic Optimization

Primal Infeasible Problems

If the problem (12.7) is infeasible (has no feasible solution), **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is the certificate of infeasibility, and the primal solution is undefined.

A certificate of primal infeasibility is a feasible solution to the modified dual problem

$$\begin{aligned} & \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\ & \text{subject to} && A^T y + s_l^x - s_u^x + s_n^x = 0, \\ & && -y + s_l^c - s_u^c = 0, \\ & && s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ & && s_n^x \in \mathcal{K}^*, \end{aligned} \quad (12.11)$$

such that the objective value is strictly positive, i.e. a solution

$$(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (s_n^x)^*)$$

to (12.11) so that

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* > 0.$$

Such a solution implies that (12.11) is unbounded, and that (12.7) is infeasible.

Dual Infeasible Problems

If the problem (12.8) is infeasible (has no feasible solution), **MOSEK** will report a certificate of dual infeasibility: The primal solution reported is the certificate of infeasibility, and the dual solution is undefined.

A certificate of dual infeasibility is a feasible solution to the modified primal problem

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && \hat{l}^c \leq Ax \leq \hat{u}^c, \\ & && \hat{l}^x \leq x \leq \hat{u}^x, \\ & && x \in K, \end{aligned} \tag{12.12}$$

where

$$\hat{l}_i^c = \begin{cases} 0 & \text{if } l_i^c > -\infty, \\ -\infty & \text{otherwise,} \end{cases} \quad \text{and} \quad \hat{u}_i^c := \begin{cases} 0 & \text{if } u_i^c < \infty, \\ \infty & \text{otherwise,} \end{cases} \tag{12.13}$$

and

$$\hat{l}_j^x = \begin{cases} 0 & \text{if } l_j^x > -\infty, \\ -\infty & \text{otherwise,} \end{cases} \quad \text{and} \quad \hat{u}_j^x := \begin{cases} 0 & \text{if } u_j^x < \infty, \\ \infty & \text{otherwise,} \end{cases} \tag{12.14}$$

such that

$$c^T x < 0.$$

Such a solution implies that (12.12) is unbounded, and that (12.8) is infeasible.

In case that both the primal problem (12.7) and the dual problem (12.8) are infeasible, **MOSEK** will report only one of the two possible certificates — which one is not defined (**MOSEK** returns the first certificate found).

12.2.3 Minimalization vs. Maximalization

When the objective sense of problem (12.7) is maximization, i.e.

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

the objective sense of the dual problem changes to minimization, and the domain of all dual variables changes sign in comparison to (12.2). The dual problem thus takes the form

$$\begin{aligned} & \text{minimize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ & \text{subject to} && A^T y + s_l^x - s_u^x + s_n^x = c, \\ & && -y + s_l^c - s_u^c = 0, \\ & && s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ & && -s_n^x \in \mathcal{K}^* \end{aligned}$$

This means that the duality gap, defined in (12.9) as the primal minus the dual objective value, becomes nonpositive. It follows that the dual objective will always be greater than or equal to the primal objective. The primal infeasibility certificate will be reported by **MOSEK** as a solution to the system

$$\begin{aligned} & A^T y + s_l^x - s_u^x + s_n^x = 0, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ & -s_n^x \in \mathcal{K}^* \end{aligned} \tag{12.15}$$

such that the objective value is strictly negative

$$(l^c)^T(s_l^c)^* - (u^c)^T(s_u^c)^* + (l^x)^T(s_l^x)^* - (u^x)^T(s_u^x)^* < 0.$$

Similarly, the certificate of dual infeasibility is an x satisfying the requirements of (12.12) such that $c^T x > 0$.

12.3 Semidefinite Optimization

Semidefinite optimization is an extension of conic optimization (see Sec. 12.2) allowing positive semidefinite matrix variables to be used in addition to the usual scalar variables. All the other parts of the input are defined exactly as in Sec. 12.2, and the discussion from that section applies verbatim to all properties of problems with semidefinite variables. We only briefly indicate how the corresponding formulae should be modified with semidefinite terms.

A semidefinite optimization problem can be written as

$$\begin{aligned} & \text{minimize} && \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \langle \bar{C}_j, \bar{X}_j \rangle + c^f \\ & \text{subject to} && \begin{aligned} l_i^c &\leq \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \langle \bar{A}_{ij}, \bar{X}_j \rangle &\leq u_i^c, & i = 0, \dots, m-1 \\ l_j^x &\leq x_j &\leq u_j^x, & j = 0, \dots, n-1 \end{aligned} \\ & && x \in \mathcal{K}, \\ & && \bar{X}_j \in \mathcal{S}_+^{r_j}, && j = 0, \dots, p-1 \end{aligned} \quad (12.16)$$

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

$$\langle U, V \rangle := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} U_{ij} V_{ij}.$$

As always we write $A = (a_{i,j})$ for the linear coefficient matrix.

Duality

The definition of the dual problem (12.8) becomes:

$$\begin{aligned} & \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f \\ & \text{subject to} && \begin{aligned} A^T y + s_l^x - s_u^x + s_n^x &= c \\ -y + s_l^c - s_u^c &= 0, \\ \bar{C}_j - \sum_{i=0}^{m-1} y_i \bar{A}_{ij} &= \bar{S}_j, && j = 0, \dots, p-1 \\ s_l^c, s_u^c, s_l^x, s_u^x &\geq 0, \\ s_n^x &\in \mathcal{K}^*, \\ \bar{S}_j &\in \mathcal{S}_+^{r_j}, && j = 0, \dots, p-1. \end{aligned} \end{aligned} \quad (12.17)$$

The duality gap (12.9) is computed as:

$$\begin{aligned} & c^T x^* + c^f - \{ (l^c)^T(s_l^c)^* - (u^c)^T(s_u^c)^* + (l^x)^T(s_l^x)^* - (u^x)^T(s_u^x)^* + c^f \} \\ &= \sum_{i=0}^{m-1} [(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*)] \\ &+ \sum_{j=0}^{n-1} [(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*)] + \sum_{j=0}^{p-1} (s_n^x)_j^* x_j^* + \sum_{j=0}^{p-1} \langle \bar{X}_j, \bar{S}_j \rangle \geq 0. \end{aligned} \quad (12.18)$$

Complementarity conditions (12.10) include the additional relation:

$$\langle \bar{X}_j, \bar{S}_j \rangle = 0 \quad j = 0, \dots, p-1. \quad (12.19)$$

Infeasibility

A certificate of primal infeasibility (12.11) is now a feasible solution to:

$$\begin{aligned}
& \text{maximize} && (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\
& \text{subject to} && A^T y + s_l^x - s_u^x + s_n^x = 0, \\
& && -y + s_l^c - s_u^c = 0, \\
& && \sum_{i=0}^{m-1} y_i \bar{A}_{ij} + \bar{S}_j = 0, && j = 0, \dots, p-1 \\
& && s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\
& && s_n^x \in \mathcal{K}^*, \\
& && \bar{S}_j \in \mathcal{S}_+^{r_j}, && j = 0, \dots, p-1.
\end{aligned} \tag{12.20}$$

such that the objective value is strictly positive.

Similarly, a dual infeasibility certificate (12.12) is a feasible solution to

$$\begin{aligned}
& \text{minimize} && \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \langle \bar{C}_j, \bar{X}_j \rangle \\
& \text{subject to} && \hat{l}_i^c \leq \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \langle \bar{A}_{ij}, \bar{X}_j \rangle \leq \hat{u}_i^c, && i = 0, \dots, m-1 \\
& && \hat{l}_j^x \leq x_j \leq \hat{u}_j^x, && j = 0, \dots, n-1 \\
& && x \in \mathcal{K}, \\
& && \bar{X}_j \in \mathcal{S}_+^{r_j}, && j = 0, \dots, p-1
\end{aligned} \tag{12.21}$$

where the modified bounds are as in (12.13) and (12.14) and the objective value is strictly negative.

Chapter 13

Optimizers

The most essential part of **MOSEK** are the optimizers:

- *primal simplex* (linear problems),
- *dual simplex* (linear problems),
- *interior-point* (linear, quadratic and conic problems),
- *mixed-integer* (problems with integer variables).

The structure of a successful optimization process is roughly:

- **Presolve**
 1. *Elimination*: Reduce the size of the problem.
 2. *Dualizer*: Choose whether to solve the primal or the dual form of the problem.
 3. *Scaling*: Scale the problem for better numerical stability.
- **Optimization**
 1. *Optimize*: Solve the problem using selected method.
 2. *Terminate*: Stop the optimization when specific termination criteria have been met.
 3. *Report*: Return the solution or an infeasibility certificate.

The preprocessing stage is transparent to the user, but useful to know about for tuning purposes. The purpose of the preprocessing steps is to make the actual optimization more efficient and robust. We discuss the details of the above steps in the following sections.

13.1 Presolve

Before an optimizer actually performs the optimization the problem is preprocessed using the so-called presolve. The purpose of the presolve is to

1. remove redundant constraints,
2. eliminate fixed variables,
3. remove linear dependencies,
4. substitute out (implied) free variables, and
5. reduce the size of the optimization problem in general.

After the presolved problem has been optimized the solution is automatically postsolved so that the returned solution is valid for the original problem. Hence, the presolve is completely transparent. For further details about the presolve phase, please see [AA95] and [AGMeszarosX96].

It is possible to fine-tune the behavior of the presolve or to turn it off entirely. If presolve consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This is done by setting the parameter *presolveUse* to *"off"*. The two most time-consuming steps of the presolve are

- the eliminator, and
- the linear dependency check.

Therefore, in some cases it is worthwhile to disable one or both of these.

Numerical issues in the presolve

During the presolve the problem is reformulated so that it hopefully solves faster. However, in rare cases the presolved problem may be harder to solve than the original problem. The presolve may also be infeasible although the original problem is not. If it is suspected that presolved problem is much harder to solve than the original, we suggest to first turn the eliminator off by setting the parameter `presolveEliminatorMaxNumTries` to 0. If that does not help, then trying to turn entire presolve off may help.

Since all computations are done in finite precision, the presolve employs some tolerances when concluding a variable is fixed or a constraint is redundant. If it happens that **MOSEK** incorrectly concludes a problem is primal or dual infeasible, then it is worthwhile to try to reduce the parameters `presolveTolX` and `presolveTolS`. However, if reducing the parameters actually helps then this should be taken as an indication that the problem is badly formulated.

Eliminator

The purpose of the eliminator is to eliminate free and implied free variables from the problem using substitution. For instance, given the constraints

$$\begin{aligned} y &= \sum_j x_j, \\ y, x &\geq 0, \end{aligned}$$

y is an implied free variable that can be substituted out of the problem, if deemed worthwhile. If the eliminator consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This can be done by setting the parameter `presolveEliminatorMaxNumTries` to 0. In rare cases the eliminator may cause that the problem becomes much hard to solve.

Linear dependency checker

The purpose of the linear dependency check is to remove linear dependencies among the linear equalities. For instance, the three linear equalities

$$\begin{aligned} x_1 + x_2 + x_3 &= 1, \\ x_1 + 0.5x_2 &= 0.5, \\ 0.5x_2 + x_3 &= 0.5. \end{aligned}$$

contain exactly one linear dependency. This implies that one of the constraints can be dropped without changing the set of feasible solutions. Removing linear dependencies is in general a good idea since it reduces the size of the problem. Moreover, the linear dependencies are likely to introduce numerical problems in the optimization phase. It is best practice to build models without linear dependencies, but that is not always easy for the user to control. If the linear dependencies are removed at the modeling stage, the linear dependency check can safely be disabled by setting the parameter `presolveLindepUse` to `"off"`.

Dualizer

All linear, conic, and convex optimization problems have an equivalent dual problem associated with them. **MOSEK** has built-in heuristics to determine if it is more efficient to solve the primal or dual problem. The form (primal or dual) is displayed in the **MOSEK** log and available as an information item from the solver. Should the internal heuristics not choose the most efficient form of the problem it may be worthwhile to set the dualizer manually by setting the parameters:

- `intpntSolveForm`: In case of the interior-point optimizer.
- `simSolveForm`: In case of the simplex optimizer.

Note that currently only linear and conic (but not semidefinite) problems may be automatically dualized.

Scaling

Problems containing data with large and/or small coefficients, say $1.0e + 9$ or $1.0e - 7$, are often hard to solve. Significant digits may be truncated in calculations with finite precision, which can result in the optimizer relying on inaccurate data. Since computers work in finite precision, extreme coefficients should be avoided. In general, data around the same *order of magnitude* is preferred, and we will refer to a problem, satisfying this loose property, as being *well-scaled*. If the problem is not well scaled, **MOSEK** will try to scale (multiply) constraints and variables by suitable constants. **MOSEK** solves the scaled problem to improve the numerical properties.

The scaling process is transparent, i.e. the solution to the original problem is reported. It is important to be aware that the optimizer terminates when the termination criterion is met on the scaled problem, therefore significant primal or dual infeasibilities may occur after unscaling for badly scaled problems. The best solution of this issue is to reformulate the problem, making it better scaled.

By default **MOSEK** heuristically chooses a suitable scaling. The scaling for interior-point and simplex optimizers can be controlled with the parameters *intpntScaling* and *simScaling* respectively.

13.2 Linear Optimization

13.2.1 Optimizer Selection

Two different types of optimizers are available for linear problems: The default is an interior-point method, and the alternative is the simplex method (primal or dual). The optimizer can be selected using the parameter *optimizer*.

The Interior-point or the Simplex Optimizer?

Given a linear optimization problem, which optimizer is the best: the simplex or the interior-point optimizer? It is impossible to provide a general answer to this question. However, the interior-point optimizer behaves more predictably: it tends to use between 20 and 100 iterations, almost independently of problem size, but cannot perform warm-start. On the other hand the simplex method can take advantage of an initial solution, but is less predictable from cold-start. The interior-point optimizer is used by default.

The Primal or the Dual Simplex Variant?

MOSEK provides both a primal and a dual simplex optimizer. Predicting which simplex optimizer is faster is impossible, however, in recent years the dual optimizer has seen several algorithmic and computational improvements, which, in our experience, make it faster on average than the primal version. Still, it depends much on the problem structure and size. Setting the *optimizer* parameter to *"freeSimplex"* instructs **MOSEK** to choose one of the simplex variants automatically.

To summarize, if you want to know which optimizer is faster for a given problem type, it is best to try all the options.

13.2.2 The Interior-point Optimizer

The purpose of this section is to provide information about the algorithm employed in the **MOSEK** interior-point optimizer for linear problems and about its termination criteria.

The homogeneous primal-dual problem

In order to keep the discussion simple it is assumed that **MOSEK** solves linear optimization problems of standard form

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && Ax = b, \\ & && x \geq 0. \end{aligned} \tag{13.1}$$

This is in fact what happens inside **MOSEK**; for efficiency reasons **MOSEK** converts the problem to standard form before solving, then converts it back to the input form when reporting the solution.

Since it is not known beforehand whether problem (13.1) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason why **MOSEK** solves the so-called homogeneous model

$$\begin{aligned} Ax - b\tau &= 0, \\ A^T y + s - c\tau &= 0, \\ -c^T x + b^T y - \kappa &= 0, \\ x, s, \tau, \kappa &\geq 0, \end{aligned} \tag{13.2}$$

where y and s correspond to the dual variables in (13.1), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.2) always has solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one. Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (13.2) satisfies

$$x_j^* s_j^* = 0 \text{ and } \tau^* \kappa^* = 0.$$

Moreover, there is always a solution that has the property $\tau^* + \kappa^* > 0$.

First, assume that $\tau^* > 0$. It follows that

$$\begin{aligned} A \frac{x^*}{\tau^*} &= b, \\ A^T \frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} &= c, \\ -c^T \frac{x^*}{\tau^*} + b^T \frac{y^*}{\tau^*} &= 0, \\ x^*, s^*, \tau^*, \kappa^* &\geq 0. \end{aligned}$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau^*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left\{ \frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*} \right\}$$

is a primal-dual optimal solution (see Sec. 12.1 for the mathematical background on duality and optimality).

On other hand, if $\kappa^* > 0$ then

$$\begin{aligned} Ax^* &= 0, \\ A^T y^* + s^* &= 0, \\ -c^T x^* + b^T y^* &= \kappa^*, \\ x^*, s^*, \tau^*, \kappa^* &\geq 0. \end{aligned}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.3}$$

or

$$b^T y^* > 0 \tag{13.4}$$

is satisfied. If (13.3) is satisfied then x^* is a certificate of dual infeasibility, whereas if (13.4) is satisfied then y^* is a certificate of primal infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm [And09].

Interior-point Termination Criterion

For efficiency reasons it is not practical to solve the homogeneous model exactly. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In the k -th iteration of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to homogeneous model is generated, where

$$x^k, s^k, \tau^k, \kappa^k > 0.$$

Optimal case

Whenever the trial solution satisfies the criterion

$$\begin{aligned} \left\| A \frac{x^k}{\tau^k} - b \right\|_{\infty} &\leq \epsilon_p (1 + \|b\|_{\infty}), \\ \left\| A^T \frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_{\infty} &\leq \epsilon_d (1 + \|c\|_{\infty}), \text{ and} \\ \min \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left| \frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k} \right| \right) &\leq \epsilon_g \max \left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k} \right), \end{aligned} \quad (13.5)$$

the interior-point optimizer is terminated and

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is reported as the primal-dual optimal solution. The interpretation of (13.5) is that the optimizer is terminated if

- $\frac{x^k}{\tau^k}$ is approximately primal feasible,
- $\left\{ \frac{y^k}{\tau^k}, \frac{s^k}{\tau^k} \right\}$ is approximately dual feasible, and
- the duality gap is almost zero.

Dual infeasibility certificate

On the other hand, if the trial solution satisfies

$$-\epsilon_i c^T x^k > \frac{\|c\|_{\infty}}{\max(1, \|b\|_{\infty})} \|Ax^k\|_{\infty}$$

then the problem is declared dual infeasible and x^k is reported as a certificate of dual infeasibility. The motivation for this stopping criterion is as follows: First assume that $\|Ax^k\|_{\infty} = 0$; then x^k is an exact certificate of dual infeasibility. Next assume that this is not the case, i.e.

$$\|Ax^k\|_{\infty} > 0,$$

and define

$$\bar{x} := \epsilon_i \frac{\max(1, \|b\|_{\infty})}{\|Ax^k\|_{\infty} \|c\|_{\infty}} x^k.$$

It is easy to verify that

$$\|A\bar{x}\|_{\infty} = \epsilon_i \frac{\max(1, \|b\|_{\infty})}{\|c\|_{\infty}} \text{ and } -c^T \bar{x} > 1,$$

which shows \bar{x} is an approximate certificate of dual infeasibility, where ϵ_i controls the quality of the approximation. A smaller value means a better approximation.

Primal infeasibility certificate

Finally, if

$$\epsilon_i b^T y^k > \frac{\|b\|_\infty}{\max(1, \|c\|_\infty)} \|A^T y^k + s^k\|_\infty$$

then y^k is reported as a certificate of primal infeasibility.

Adjusting optimality criteria

It is possible to adjust the tolerances ϵ_p , ϵ_d , ϵ_g and ϵ_i using parameters; see table for details.

Table 13.1: Parameters employed in termination criterion

ToleranceParameter	name
ϵ_p	<i>intpntTolPfeas</i>
ϵ_d	<i>intpntTolDfeas</i>
ϵ_g	<i>intpntTolRelGap</i>
ϵ_i	<i>intpntTolInfeas</i>

The default values of the termination tolerances are chosen such that for a majority of problems appearing in practice it is not possible to achieve much better accuracy. Therefore, tightening the tolerances usually is not worthwhile. However, an inspection of (13.5) reveals that the quality of the solution depends on $\|b\|_\infty$ and $\|c\|_\infty$; the smaller the norms are, the better the solution accuracy.

The interior-point method as implemented by **MOSEK** will converge toward optimality and primal and dual feasibility at the same rate [And09]. This means that if the optimizer is stopped prematurely then it is very unlikely that either the primal or dual solution is feasible. Another consequence is that in most cases all the tolerances, ϵ_p , ϵ_d , ϵ_g and ϵ_i , have to be relaxed together to achieve an effect.

The basis identification discussed in Sec. 13.2.2 requires an optimal solution to work well; hence basis identification should be turned off if the termination criterion is relaxed.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

Basis Identification

An interior-point optimizer does not return an optimal basic solution unless the problem has a unique primal and dual optimal solution. Therefore, the interior-point optimizer has an optional post-processing step that computes an optimal basic solution starting from the optimal interior-point solution. More information about the basis identification procedure may be found in [AY96]. In the following we provide an overall idea of the procedure.

There are some cases in which a basic solution could be more valuable:

- a basic solution is often more accurate than an interior-point solution,
- a basic solution can be used to warm-start the simplex algorithm in case of reoptimization,
- a basic solution is in general more sparse, i.e. more variables are fixed to zero. This is particularly appealing when solving continuous relaxations of mixed integer problems, as well as in all applications in which sparser solutions are preferred.

To illustrate how the basis identification routine works, we use the following trivial example:

$$\begin{aligned} \text{minimize} \quad & x + y \\ \text{subject to} \quad & x + y = 1, \\ & x, y \geq 0. \end{aligned}$$

It is easy to see that all feasible solutions are also optimal. In particular, there are two basic solutions, namely

$$\begin{aligned} (x_1^*, y_1^*) &= (1, 0), \\ (x_2^*, y_2^*) &= (0, 1). \end{aligned}$$

The interior point algorithm will actually converge to the center of the optimal set, i.e. to $(x^*, y^*) = (1/2, 1/2)$ (to see this in **MOSEK** deactivate *Presolve*).

- **DFEAS**: $\|A^T y^k + s^k - c\tau^k\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- **GFEAS**: $|-c^T x^k + b^T y^k - \kappa^k|$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- **PRSTATUS**: This number converges to 1 if the problem has an optimal solution whereas it converges to -1 if that is not the case.
- **POBJ**: $c^T x^k / \tau^k$. An estimate for the primal objective value.
- **DOBJ**: $b^T y^k / \tau^k$. An estimate for the dual objective value.
- **MU**: $\frac{(x^k)^T s^k + \tau^k \kappa^k}{n+1}$. The numbers in this column should always converge to zero.
- **TIME**: Time spent since the optimization started.

13.2.3 The Simplex Optimizer

An alternative to the interior-point optimizer is the simplex optimizer. The simplex optimizer uses a different method that allows exploiting an initial guess for the optimal solution to reduce the solution time. Depending on the problem it may be faster or slower to use an initial guess; see [Sec. 13.2.1](#) for a discussion. **MOSEK** provides both a primal and a dual variant of the simplex optimizer.

Simplex Termination Criterion

The simplex optimizer terminates when it finds an optimal basic solution or an infeasibility certificate. A basic solution is optimal when it is primal and dual feasible; see [Sec. 12.1](#) for a definition of the primal and dual problem. Due to the fact that computations are performed in finite precision **MOSEK** allows violations of primal and dual feasibility within certain tolerances. The user can control the allowed primal and dual tolerances with the parameters `basisTolX` and `basisTolS`.

Setting the parameter `optimizer` to `"freeSimplex"` instructs **MOSEK** to select automatically between the primal and the dual simplex optimizers. Hence, **MOSEK** tries to choose the best optimizer for the given problem and the available solution. The same parameter can also be used to force one of the variants.

Starting From an Existing Solution

When using the simplex optimizer it may be possible to reuse an existing solution and thereby reduce the solution time significantly. When a simplex optimizer starts from an existing solution it is said to perform a *warm-start*. If the user is solving a sequence of optimization problems by solving the problem, making modifications, and solving again, **MOSEK** will warm-start automatically.

By default **MOSEK** uses presolve when performing a warm-start. If the optimizer only needs very few iterations to find the optimal solution it may be better to turn off the presolve.

Numerical Difficulties in the Simplex Optimizers

Though **MOSEK** is designed to minimize numerical instability, completely avoiding it is impossible when working in finite precision. **MOSEK** treats a “numerically unexpected behavior” event inside the optimizer as a *set-back*. The user can define how many set-backs the optimizer accepts; if that number is exceeded, the optimization will be aborted. Set-backs are a way to escape long sequences where the optimizer tries to recover from an unstable situation.

Examples of set-backs are: repeated singularities when factorizing the basis matrix, repeated loss of feasibility, degeneracy problems (no progress in objective) and other events indicating numerical difficulties. If the simplex optimizer encounters a lot of set-backs the problem is usually badly scaled; in such a situation try to reformulate it into a better scaled problem. Then, if a lot of set-backs still occur, trying one or more of the following suggestions may be worthwhile:

- Raise tolerances for allowed primal or dual feasibility: increase the value of
 - `basisTolX`, and
 - `basisTolS`.

- Raise or lower pivot tolerance: Change the *simplexAbsTolPiv* parameter.
- Switch optimizer: Try another optimizer.
- Switch off crash: Set both *simPrimalCrash* and *simDualCrash* to 0.
- Experiment with other pricing strategies: Try different values for the parameters
 - *simPrimalSelection* and
 - *simDualSelection*.
- If you are using warm-starts, in rare cases switching off this feature may improve stability. This is controlled by the *simHotstart* parameter.
- Increase maximum number of set-backs allowed controlled by *simMaxNumSetbacks*.
- If the problem repeatedly becomes infeasible try switching off the special degeneracy handling. See the parameter *simDegen* for details.

The Simplex Log

Below is a typical log output from the simplex optimizer:

Optimizer	- solved problem	:	the primal				
Optimizer	- Constraints	:	667				
Optimizer	- Scalar variables	:	1424		conic	:	0
Optimizer	- hotstart	:	no				
ITER	DEGITER(%)	PFEAS	DFEAS	POBJ	DOBJ		
↪	TIME	TOTTIME					
0	0.00	1.43e+05	NA	6.5584140832e+03	NA		⌋
↪	0.00	0.02					⌋
1000	1.10	0.00e+00	NA	1.4588289726e+04	NA		⌋
↪	0.13	0.14					⌋
2000	0.75	0.00e+00	NA	7.3705564855e+03	NA		⌋
↪	0.21	0.22					⌋
3000	0.67	0.00e+00	NA	6.0509727712e+03	NA		⌋
↪	0.29	0.31					⌋
4000	0.52	0.00e+00	NA	5.5771203906e+03	NA		⌋
↪	0.38	0.39					⌋
4533	0.49	0.00e+00	NA	5.5018458883e+03	NA		⌋
↪	0.42	0.44					⌋

The first lines summarize the problem the optimizer is solving. This is followed by the iteration log, with the following meaning:

- ITER: Number of iterations.
- DEGITER(%): Ratio of degenerate iterations.
- PFEAS: Primal feasibility measure reported by the simplex optimizer. The numbers should be 0 if the problem is primal feasible (when the primal variant is used).
- DFEAS: Dual feasibility measure reported by the simplex optimizer. The number should be 0 if the problem is dual feasible (when the dual variant is used).
- POBJ: An estimate for the primal objective value (when the primal variant is used).
- DOBJ: An estimate for the dual objective value (when the dual variant is used).
- TIME: Time spent since this instance of the simplex optimizer was invoked (in seconds).
- TOTTIME: Time spent since optimization started (in seconds).

13.3 Conic Optimization - Interior-point optimizer

For conic optimization problems only an interior-point type optimizer is available.

13.3.1 The homogeneous primal-dual problem

The interior-point optimizer is an implementation of the so-called homogeneous and self-dual algorithm. For a detailed description of the algorithm, please see [ART03]. In order to keep our discussion simple we will assume that **MOSEK** solves a conic optimization problem of the form:

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && Ax = b, \\ & && x \in \mathcal{K} \end{aligned} \tag{13.6}$$

where \mathcal{K} is a convex cone. The corresponding dual problem is

$$\begin{aligned} & \text{maximize} && b^T y \\ & \text{subject to} && A^T y + s = c, \\ & && s \in \mathcal{K}^* \end{aligned} \tag{13.7}$$

where \mathcal{K}^* is the dual cone of \mathcal{K} . See Sec. 12.2 for definitions.

Since it is not known beforehand whether problem (13.6) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason that **MOSEK** solves the so-called homogeneous model

$$\begin{aligned} Ax - b\tau &= 0, \\ A^T y + s - c\tau &= 0, \\ -c^T x + b^T y - \kappa &= 0, \\ x &\in \mathcal{K}, \\ s &\in \mathcal{K}^*, \\ \tau, \kappa &\geq 0, \end{aligned} \tag{13.8}$$

where y and s correspond to the dual variables in (13.6), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.8) always has a solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one. Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (13.8) satisfies

$$(x^*)^T s^* + \tau^* \kappa^* = 0$$

i.e. complementarity. Observe that $x^* \in \mathcal{K}$ and $s^* \in \mathcal{K}^*$ implies

$$(x^*)^T s^* \geq 0$$

and therefore

$$\tau^* \kappa^* = 0.$$

since $\tau^*, \kappa^* \geq 0$. Hence, at least one of τ^* and κ^* is zero.

First, assume that $\tau^* > 0$ and hence $\kappa^* = 0$. It follows that

$$\begin{aligned} A \frac{x^*}{\tau^*} &= b, \\ A^T \frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} &= c, \\ -c^T \frac{x^*}{\tau^*} + b^T \frac{y^*}{\tau^*} &= 0, \\ x^*/\tau^* &\in \mathcal{K}, \\ s^*/\tau^* &\in \mathcal{K}^*. \end{aligned}$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau^*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left(\frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*} \right)$$

is a primal-dual optimal solution.

On other hand, if $\kappa^* > 0$ then

$$\begin{aligned} Ax^* &= 0, \\ A^T y^* + s^* &= 0, \\ -c^T x^* + b^T y^* &= \kappa^*, \\ x^* &\in \mathcal{K}, \\ s^* &\in \mathcal{K}^*. \end{aligned}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.9}$$

or

$$b^T y^* > 0 \tag{13.10}$$

holds. If (13.9) is satisfied, then x^* is a certificate of dual infeasibility, whereas if (13.10) holds then y^* is a certificate of primal infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm [And09].

13.3.2 Interior-point Termination Criterion

Since computations are performed in finite precision, and for efficiency reasons, it is not possible to solve the homogeneous model exactly in general. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In every iteration k of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to the homogeneous model is generated, where

$$x^k \in \mathcal{K}, s^k \in \mathcal{K}^*, \tau^k, \kappa^k > 0.$$

Therefore, it is possible to compute the values:

$$\begin{aligned} \rho_p^k &= \arg \min_{\rho} \left\{ \rho \mid \left\| A \frac{x^k}{\tau^k} - b \right\|_{\infty} \leq \rho \varepsilon_p (1 + \|b\|_{\infty}) \right\}, \\ \rho_d^k &= \arg \min_{\rho} \left\{ \rho \mid \left\| A^T \frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_{\infty} \leq \rho \varepsilon_d (1 + \|c\|_{\infty}) \right\}, \\ \rho_g^k &= \arg \min_{\rho} \left\{ \rho \mid \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left| \frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k} \right| \right) \leq \rho \varepsilon_g \max \left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k} \right) \right\}, \\ \rho_{pi}^k &= \arg \min_{\rho} \left\{ \rho \mid \left\| A^T y^k + s^k \right\|_{\infty} \leq \rho \varepsilon_i b^T y^k, b^T y^k > 0 \right\} \text{ and} \\ \rho_{di}^k &= \arg \min_{\rho} \left\{ \rho \mid \left\| A x^k \right\|_{\infty} \leq -\rho \varepsilon_i c^T x^k, c^T x^k < 0 \right\}. \end{aligned}$$

Note $\varepsilon_p, \varepsilon_d, \varepsilon_g$ and ε_i are nonnegative user specified tolerances.

Optimal Case

Observe ρ_p^k measures how far x^k/τ^k is from being a good approximate primal feasible solution. Indeed if $\rho_p^k \leq 1$, then

$$\left\| A \frac{x^k}{\tau^k} - b \right\|_{\infty} \leq \varepsilon_p (1 + \|b\|_{\infty}). \tag{13.11}$$

This shows the violations in the primal equality constraints for the solution x^k/τ^k is small compared to the size of b given ε_p is small.

Similarly, if $\rho_d^k \leq 1$, then $(y^k, s^k)/\tau^k$ is an approximate dual feasible solution. If in addition $\rho_g \leq 1$, then the solution $(x^k, y^k, s^k)/\tau^k$ is approximate optimal because the associated primal and dual objective values are almost identical.

In other words if $\max(\rho_p^k, \rho_d^k, \rho_g^k) \leq 1$, then

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is an approximate optimal solution.

Dual Infeasibility Certificate

Next assume that $\rho_{di}^k \leq 1$ and hence

$$\|Ax^k\|_\infty \leq -\varepsilon_i c^T x^k \text{ and } -c^T x^k > 0$$

holds. Now in this case the problem is declared dual infeasible and x^k is reported as a certificate of dual infeasibility. The motivation for this stopping criterion is as follows. Let

$$\bar{x} := \frac{x^k}{-c^T x^k}$$

and it is easy to verify that

$$\|A\bar{x}\|_\infty \leq \varepsilon_i \text{ and } c^T \bar{x} = -1$$

which shows \bar{x} is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

Primal Infeasibility Certificate

Next assume that $\rho_{pi}^k \leq 1$ and hence

$$\|A^T y^k + s^k\|_\infty \leq \varepsilon_i b^T y^k \text{ and } b^T y^k > 0$$

holds. Now in this case the problem is declared primal infeasible and (y^k, s^k) is reported as a certificate of primal infeasibility. The motivation for this stopping criterion is as follows. Let

$$\bar{y} := \frac{y^k}{b^T y^k} \text{ and } \bar{s} := \frac{s^k}{b^T y^k}$$

and it is easy to verify that

$$\|A^T \bar{y} + \bar{s}\|_\infty \leq \varepsilon_i \text{ and } b^T \bar{y} = 1$$

which shows (\bar{y}, \bar{s}) is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

13.3.3 Adjusting optimality criteria

It is possible to adjust the tolerances ε_p , ε_d , ε_g and ε_i using parameters; see table for details.

Table 13.2: Parameters employed in termination criterion

ToleranceParameter	name
ε_p	<i>intpntCoTolPfeas</i>
ε_d	<i>intpntCoTolDfeas</i>
ε_g	<i>intpntCoTolRelGap</i>
ε_i	<i>intpntCoTolInfeas</i>

The default values of the termination tolerances are chosen such that for a majority of problems appearing in practice it is not possible to achieve much better accuracy. Therefore, tightening the tolerances usually is not worthwhile. However, an inspection of (13.11) reveals that the quality of the solution depends on $\|b\|_\infty$ and $\|c\|_\infty$; the smaller the norms are, the better the solution accuracy.

The interior-point method as implemented by **MOSEK** will converge toward optimality and primal and dual feasibility at the same rate [And09]. This means that if the optimizer is stopped prematurely then it is very unlikely that either the primal or dual solution is feasible. Another consequence is that in most cases all the tolerances, ε_p , ε_d , ε_g and ε_i , have to be relaxed together to achieve an effect.

If the optimizer terminates without locating a solution that satisfies the termination criteria, for example because of a stall or other numerical issues, then it will check if the solution found up to that point satisfies the same criteria with all tolerances multiplied by the value of `intpntCoTolNearRel`. If this is the case, the solution is still declared as optimal.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

13.3.4 The Interior-point Log

Below is a typical log output from the interior-point optimizer:

Optimizer	- threads	:	20					
Optimizer	- solved problem	:	the primal					
Optimizer	- Constraints	:	1					
Optimizer	- Cones	:	2					
Optimizer	- Scalar variables	:	6	conic	:	6		
Optimizer	- Semi-definite variables:	0		scalarized	:	0		
Factor	- setup time	:	0.00	dense det. time	:	0.00		
Factor	- ML order time	:	0.00	GP order time	:	0.00		
Factor	- nonzeros before factor	:	1	after factor	:	1		
Factor	- dense dim.	:	0	flops	:	1.		
↪ 70e+01								
ITE	PFEAS	DFEAS	GFEAS	PRSTATUS	POBJ	DOBJ	MU	▬
↪ TIME								
0	1.0e+00	2.9e-01	3.4e+00	0.00e+00	2.414213562e+00	0.000000000e+00	1.0e+00	▬
↪ 0.01								
1	2.7e-01	7.9e-02	2.2e+00	8.83e-01	6.969257574e-01	-9.685901771e-03	2.7e-01	▬
↪ 0.01								
2	6.5e-02	1.9e-02	1.2e+00	1.16e+00	7.606090061e-01	6.046141322e-01	6.5e-02	▬
↪ 0.01								
3	1.7e-03	5.0e-04	2.2e-01	1.12e+00	7.084385672e-01	7.045122560e-01	1.7e-03	▬
↪ 0.01								
4	1.4e-08	4.2e-09	4.9e-08	1.00e+00	7.071067941e-01	7.071067599e-01	1.4e-08	▬
↪ 0.01								

The first line displays the number of threads used by the optimizer and the second line indicates if the optimizer chose to solve the primal or dual problem (see `intpntSolveForm`). The next lines display the problem dimensions as seen by the optimizer, and the **Factor...** lines show various statistics. This is followed by the iteration log.

Using the same notation as in Sec. 13.3.1 the columns of the iteration log have the following meaning:

- **ITE**: Iteration index k .
- **PFEAS**: $\|Ax^k - b\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- **DFEAS**: $\|A^T y^k + s^k - c\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- **GFEAS**: $|-c^T x^k + b^T y^k - \kappa^k|$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- **PRSTATUS**: This number converges to 1 if the problem has an optimal solution whereas it converges to -1 if that is not the case.

- **POBJ:** $c^T x^k / \tau^k$. An estimate for the primal objective value.
- **DOBJ:** $b^T y^k / \tau^k$. An estimate for the dual objective value.
- **MU:** $\frac{(x^k)^T s^k + \tau^k \kappa^k}{n+1}$. The numbers in this column should always converge to zero.
- **TIME:** Time spent since the optimization started (in seconds).

13.4 The Optimizer for Mixed-integer Problems

A problem is a mixed-integer optimization problem when one or more of the variables are constrained to be integer valued. Readers unfamiliar with integer optimization are recommended to consult some relevant literature, e.g. the book [Wol98] by Wolsey.

13.4.1 The Mixed-integer Optimizer Overview

MOSEK can solve mixed-integer linear and conic problems, except for mixed-integer semidefinite problems.

By default the mixed-integer optimizer is run-to-run deterministic. This means that if a problem is solved twice on the same computer with identical parameter settings and no time limit then the obtained solutions will be identical. If a time limit is set then this may not be case since the time taken to solve a problem is not deterministic. The mixed-integer optimizer is parallelized i.e. it can exploit multiple cores during the optimization.

The solution process can be split into these phases:

1. **Presolve:** See [Sec. 13.1](#).
2. **Cut generation:** Valid inequalities (cuts) are added to improve the lower bound.
3. **Heuristic:** Using heuristics the optimizer tries to guess a good feasible solution. Heuristics can be controlled by the parameter *mioHeuristicLevel*.
4. **Search:** The optimal solution is located by branching on integer variables.

13.4.2 Relaxations and bounds

It is important to understand that, in a worst-case scenario, the time required to solve integer optimization problems grows exponentially with the size of the problem (solving mixed-integer problems is NP-hard). For instance, a problem with n binary variables, may require time proportional to 2^n . The value of 2^n is huge even for moderate values of n .

In practice this implies that the focus should be on computing a near-optimal solution quickly rather than on locating an optimal solution. Even if the problem is only solved approximately, it is important to know how far the approximate solution is from an optimal one. In order to say something about the quality of an approximate solution the concept of *relaxation* is important.

Consider for example a mixed-integer optimization problem

$$\begin{aligned} z^* = \quad & \text{minimize} && c^T x \\ & \text{subject to} && Ax = b, \\ & && x \geq 0 \\ & && x_j \in \mathbb{Z}, \quad \forall j \in \mathcal{J}. \end{aligned} \tag{13.12}$$

It has the continuous relaxation

$$\begin{aligned} \underline{z} = \quad & \text{minimize} && c^T x \\ & \text{subject to} && Ax = b, \\ & && x \geq 0 \end{aligned} \tag{13.13}$$

obtained simply by ignoring the integrality restrictions. The relaxation is a continuous problem, and therefore much faster to solve to optimality with a linear (or, in the general case, conic) optimizer.

We call the optimal value \underline{z} the *objective bound*. The objective bound \underline{z} normally increases during the solution search process when the continuous relaxation is gradually refined.

Moreover, if \hat{x} is any feasible solution to (13.12) and

$$\bar{z} := c^T \hat{x}$$

then

$$\underline{z} \leq z^* \leq \bar{z}.$$

These two inequalities allow us to estimate the quality of the integer solution: it is no further away from the optimum than $\bar{z} - \underline{z}$ in terms of the objective value. Whenever a mixed-integer problem is solved **MOSEK** reports this lower bound so that the quality of the reported solution can be evaluated.

13.4.3 Outer approximation for mixed-integer conic problems

The relaxations of mixed integer conic problems can be solved either as a nonlinear problem with the interior point algorithm (default) or with a linear outer approximation algorithm. The type of relaxation used can be set with `mioConicOuterApproximation`. The best value for this option is highly problem dependent.

13.4.4 Randomization

A number of internal algorithms of the mixed-integer solver are dependent on random tie-breaking. The random tie-breaking can have a significant impact on the path taken by the algorithm and the optimal solution returned. The random seed can be set with the parameter `mioSeed`.

13.4.5 Termination Criterion

In general, it is time consuming to find an exact feasible and optimal solution to an integer optimization problem, though in many practical cases it may be possible to find a sufficiently good solution. The issue of terminating the mixed-integer optimizer is rather delicate and the user has numerous possibilities of influencing it with various parameters. The mixed-integer optimizer employs a relaxed feasibility and optimality criterion to determine when a satisfactory solution is located.

A candidate solution that is feasible for the continuous relaxation is said to be an *integer feasible solution* if the criterion

$$\min(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j) \leq \delta_1 \quad \forall j \in \mathcal{J}$$

is satisfied, meaning that x_j is at most δ_1 from the nearest integer.

Whenever the integer optimizer locates an integer feasible solution it will check if the criterion

$$\bar{z} - \underline{z} \leq \max(\delta_2, \delta_3 \max(\delta_4, |\bar{z}|))$$

is satisfied. If this is the case, the integer optimizer terminates and reports the integer feasible solution as an optimal solution.

All the δ tolerances discussed above can be adjusted using suitable parameters — see Table 13.3.

Table 13.3: Tolerances for the mixed-integer optimizer.

Tolerance	Parameter name
δ_1	<code>mioTolAbsRelaxInt</code>
δ_2	<code>mioTolAbsGap</code>
δ_3	<code>mioTolRelGap</code>
δ_4	<code>mioRelGapConst</code>

In Table 13.4 some other common parameters affecting the integer optimizer termination criterion are shown.

Table 13.4: Other parameters affecting the integer optimizer termination criterion.

Parameter name	Explanation
<i>mioMaxNumBranches</i>	Maximum number of branches allowed.
<i>mioMaxNumRelaxs</i>	Maximum number of relaxations allowed.
<i>mioMaxNumSolutions</i>	Maximum number of feasible integer solutions allowed.

13.4.6 Speeding Up the Solution Process

As mentioned previously, in many cases it is not possible to find an optimal solution to an integer optimization problem in a reasonable amount of time. Some suggestions to reduce the solution time are:

- Relax the termination criterion: In case the run time is not acceptable, the first thing to do is to relax the termination criterion — see [Sec. 13.4.5](#) for details.
- Specify a good initial solution: In many cases a good feasible solution is either known or easily computed using problem-specific knowledge. If a good feasible solution is known, it is usually worthwhile to use this as a starting point for the integer optimizer. See [Sec. 7.6.2](#).
- Improve the formulation: A mixed-integer optimization problem may be impossible to solve in one form and quite easy in another form. However, it is beyond the scope of this manual to discuss good formulations for mixed-integer problems. For discussions on this topic see for example [\[Wol98\]](#).

13.4.7 Understanding Solution Quality

To determine the quality of the solution one should check the following:

- The problem status and solution status returned by **MOSEK**, as well as constraint violations in case of suboptimal solutions.
- The *optimality gap* defined as

$$\epsilon = |(\text{objective value of feasible solution}) - (\text{objective bound})| = |\bar{z} - \underline{z}|.$$

which measures how much the located solution can deviate from the optimal solution to the problem. The optimality gap can be retrieved through the information item *"mioObjAbsGap"*. Often it is more meaningful to look at the relative optimality gap normalized against the magnitude of the solution.

$$\epsilon_{\text{rel}} = \frac{|\bar{z} - \underline{z}|}{\max(\delta_4, |\bar{z}|)}.$$

The relative optimality gap is available in the information item *"mioObjRelGap"*.

13.4.8 The Mixed-integer Log

Below is a typical log output from the mixed-integer optimizer:

Presolved problem: 6573 variables, 35728 constraints, 101258 non-zeros						
Presolved problem: 0 general integer, 4294 binary, 2279 continuous						
Clique table size: 1636						
BRANCHES	RELAXS	ACT_NDS	DEPTH	BEST_INT_OBJ	BEST_RELAX_OBJ	REL_GAP (
↪%)	TIME					
0	1	0	0	NA	1.8218819866e+07	NA
↪	1.6					
0	1	0	0	1.8331557950e+07	1.8218819866e+07	0.61
↪	3.5					
0	1	0	0	1.8300507546e+07	1.8218819866e+07	0.45
↪	4.3					

(continues on next page)

(continued from previous page)

Cut generation started.							
0	2	0	0	1.8300507546e+07	1.8218819866e+07	0.45	┐
↪	5.3						
Cut generation terminated. Time = 1.43							
0	3	0	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	7.5						
15	18	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	10.5						
31	34	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	11.1						
51	54	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	11.6						
91	94	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	12.4						
171	174	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	14.3						
331	334	1	0	1.8286893047e+07	1.8231580587e+07	0.30	┐
↪	17.9						
[...]							
Objective of best integer solution : 1.825846762609e+07							
Best objective bound : 1.823311032986e+07							
Construct solution objective : Not employed							
Construct solution # roundings : 0							
User objective cut value : 0							
Number of cuts generated : 117							
Number of Gomory cuts : 108							
Number of CMIR cuts : 9							
Number of branches : 4425							
Number of relaxations solved : 4410							
Number of interior point iterations: 25							
Number of simplex iterations : 221131							

The first lines contain a summary of the problem as seen by the optimizer. This is followed by the iteration log. The columns have the following meaning:

- BRANCHES: Number of branches generated.
- RELAXS: Number of relaxations solved.
- ACT_NDS: Number of active branch bound nodes.
- DEPTH: Depth of the recently solved node.
- BEST_INT_OBJ: The best integer objective value, \bar{z} .
- BEST_RELAX_OBJ: The best objective bound, \underline{z} .
- REL_GAP(%): Relative optimality gap, $100\% \cdot \epsilon_{\text{rel}}$
- TIME: Time (in seconds) from the start of optimization.

Following that a summary of the optimization process is printed.

Chapter 14

Fusion API Reference

- *General API conventions.*
- **mosek::fusion** classes
 - Quick links: *Model*, *Expr*, *Variable*, *Parameter*, *Var*, *Domain*, *Matrix*, *Constraint*
 - *Full list*
- **Optimizer parameters:**
 - *Double*, *Integer*, *String*
 - *Full list*
 - *Browse by topic*
- **Optimizer information items:**
 - *Double*, *Integer*, *Long*
- *Enumerations*
- *Constants*
- *Exceptions*
- *Linear algebra utilities*

14.1 *Fusion* API conventions

14.1.1 General conventions

All the classes of the *Fusion* interface are contained in the namespace **mosek::fusion**. The user should not directly instantiate objects of any class other than creating a *Model*.

```
Model::t M = new Model();    auto _M = finally([&]() { M->dispose(); });
```

The model is the main access point to an optimization problem and its solution. All other objects should be created through the model (*Model.variable*, *Model.constraint*, etc.) or using static factory methods (*Matrix.sparse* etc.).

The C++ *Fusion* implements its own reference counting and array types to improve garbage collection. The user will require the type definitions from two namespaces:

```
#include "fusion.h"

using namespace mosek::fusion;
using namespace monty;
```

The following subsections contain the relevant API reference.

14.1.2 C++ Arrays and Pointers

Arrays

All arrays in *Fusion* are passed as objects of type `monty::ndarray` wrapped in a shared pointer:

```
std::shared_ptr<monty::ndarray<T,N>>
```

where `T` is the type of the elements and `N` is an integer denoting the number of dimensions. See `monty::ndarray` for ways to create, manipulate and iterate over arrays and `monty::new_array_ptr` for array factory methods.

Shapes and indexes

The shape of arrays is defined by a type `monty::shape_t`, which is also used to define an N -dimensional index.

Objects and reference counting

Objects of all classes defined in the `mosek::fusion` namespace are wrapped in a reference counting pointer of type `monty::rc_ptr`. Every *Fusion* class `C` contains a definition of a type

```
typedef monty::rc_ptr<C> t;
```

for example `Model::t`, `Variable::t` etc. These pointer types should be used instead of standard pointers to ensure proper garbage collection. It is also recommended to use a finalizer if possible, as below. For example

```
Model::t M = new Model("MyModel");
auto _M = finally([&]() { M->dispose(); });
Variable::t v = M->variable(5);
```

An object will be destroyed the moment nobody refers to it any more.

14.1.3 Using vectors

Standard C++ vectors can be converted into a *Fusion* array using `monty::new_array_ptr`.

```
std::vector<double> x(5);
// ....
auto a = monty::new_array_ptr<double>(x);
// Now we can use a in Fusion calls, eg:
auto product = Expr::dot(a, v);
```

Similar version of `monty::new_array_ptr` is available for 2D vectors:

```
std::vector<std::vector<double>> x({{1,1,1,1,1}, {1,2,3,4,5}});
// ....
auto product = Expr::mul(monty::new_array_ptr<double>(x), v);
```

For more details see `monty::new_array_ptr`, and `monty::new_vector_from_array_ptr`.

14.1.4 C++ extension reference

Multidimensional arrays

`monty::ndarray<T,N>`

A template class for all arrays in *Fusion*.

Template parameters

- `T` – The type of objects stored in the array.
- `N` – The number of dimensions.

`ndarray<T,N>.ndarray<T,N>`

```
ndarray(shape_t<N> shape)
ndarray(shape_t<N> shape, T value)
ndarray(const shape_t<N> & shape, const std::function<T(ptrdiff_t)> & fLin);
ndarray(const shape_t<N> & shape, const std::function<T(const shape_t<N> &)> & fInd)
ndarray(T* vals, shape_t<N> shape)
ndarray(init_t & init)
template<typename Iterator> ndarray(const shape_t<N> & shape, Iterator begin,
Iterator end)
template<typename Iterable> ndarray(const shape_t<N> & shape, const typename
const_iterable_t<Iterable>::t & that)
template<typename Iterable> ndarray(const shape_t<N> & shape, const const_
iterable_t<Iterable> & that)
ndarray(const ndarray<T,N> & that)
ndarray(ndarray<T,N> && that)
```

Constructor of a multidimensional array object. The initializing data can have several forms:

- Default value, e.g. 0 for numerical types.
- Create a new array initialized with values generated by a generator function.

```
auto a1 = std::make_shared<ndarray<int,1>>(shape(5), [](ptrdiff_t i) {
return 5-i; });
auto a2 = std::make_shared<ndarray<int,2>>(shape(5,5), std::function
<int(const shape_t<2> &)>([](const shape_t<2> & p) { return p[0]+p[1]; }));
```

- Initialized with values from another array.
- Initialized with an initializer list.

Arrays can also be constructed with the factory method `monty::new_array_ptr`.

Parameters

- `shape` (*shape_t*) – The shape of the array.
- `value` (T) – The default value for all elements.
- `fLin` – An array-filling function taking the linear index as an argument.
- `fInd` – An array-filling function taking the multidimensional index as an argument.
- `vals` (T*) – An array with element values.
- `init` (*init_t*) – An initializer list. The type *init_t* is an *N*-fold nested initializer list of type `std::initializer_list`. For example, for 2-dimensional arrays this is an initializer list of initializer lists, and so forth.
- `begin`, `end` – The pointers to an iterator.
- `that` – Another iterable object (see `monty_base.h`) or an `ndarray` object used as initializer.

`ndarray<T,N>.operator()`

```
T& operator()(int i1, ..., int iN)
```

N-dimensional access operator. Returns the element at position (i_1, \dots, i_N) .

Example:

```
auto a = std::make_shared<ndarray<int,2>>(shape(5,5));
(*a)(3,4) = 15;
```

Parameters i_1, \dots, i_N (int) – The multi-dimensional index of the element.

`ndarray<T,N>.operator[]`

```
T& operator[] (int i)
T& operator[] (const shape_t<N> & idx)
```

Access operator. Returns a specific element in the array.

Parameters

- i (int) – The linear index of the element, referring to its position in the one-dimensional row-wise flattening of the array.
- idx (*shape_t*) – The multi-dimensional index of the element.

`ndarray<T,N>.begin`

```
T* begin()
```

An iterator pointing to the beginning of the array. Note that the iterator sees the array as a one dimensional array obtained traversing the N -dimensional array row-wise.

Example:

```
auto a = new_array_ptr<double,2>({ { 1.1, 2.2 }, { 3.3, 4.4 } } );
for(auto x : *a) std::cout << x << " ";
```

`ndarray<T,N>.end`

```
T* end()
```

An iterator pointing to the end of the array.

`ndarray<T,N>.raw`

```
T* raw()
```

Returns a pointer to the raw data array.

`ndarray<T,N>.size`

```
size_t size()
```

Returns the number of elements in the array.

Shapes and indices

`monty::shape_t<N>`

Represents the shape of an N -dimensional array. It is in fact an N -tuple of integers. It is also used to specify an index in an N -dimensional array.

Template parameters N – The number of dimensions.

`shape_t<N>.shape_t<N>`

```
shape_t(int i1, ..., int iN)
shape_t(int i0, shape_t<N-1> is)
shape_t(const shape_t<N> & that)
```

Constructor of a new shape. Shapes can also be constructed with the factory method `monty::shape`.

Parameters

- `i1, ..., iN (int)` – The dimensions.
- `i0 (int)` – The first dimension.
- `is (shape_t)` – Shape of the remaining $N - 1$ dimensions.
- `that (shape_t)` – Another shape to be copied.

`shape_t<N>.operator[]`

```
int operator[] (int i)
```

Return the size in a specific dimension.

Parameters `i (int)` – The dimension.

`shape_t<N>.tolinear`

```
int tolinear(const shape_t<N> & point)
int tolinear(int i1, ..., int iN)
```

Compute the linear index of an element within this shape. The linear index is the position of the element in the one-dimensional row-wise flattening of the shape.

Example:

```
std::cout << shape(2,3).tolinear(1,1);
// 4
```

Parameters

- `point (shape_t)` – The coordinates of an index within the current shape.
- `i1, ..., iN (int)` – The coordinates of an index within the current shape.

`shape_t<N>.begin`

```
shape_iterator<N> begin()
```

An iterator to the shape. This iterator will traverse all indices belonging to the shape in increasing linear order, i.e. in the row-wise order in the multidimensional shape, and return a object of type *shape_t* for each index.

Example:

```
for(auto p : shape(2,3)) std::cout << p;
// (0,0) (0,1) (0,2) (1,0) (1,1) (1,2)
```

`shape_t<N>.end`

```
shape_iterator<N> end()
```

An iterator pointing to the end of shape.

Reference counting pointers

`monty::rc_ptr<T>`

A reference counting pointer wrapped around an object of type `T`. Implements standard pointer type operators such as `*` and `->` in the expected way. For *Fusion* classes, the reference counting pointer type `monty::rc_ptr<C>` is defined as `C::t`.

Exceptions

`monty::ArrayInitializerException`

An exception thrown when the array initializer is incorrect, for instance a 2-dimensional array is initialized with row lists of non-equal lengths.

Auxiliary functions in the `monty` namespace

`monty::new_array_ptr`

```
std::shared_ptr<ndarray<T,N>> new_array_ptr(init_t init)
std::shared_ptr<ndarray<T,1>> new_array_ptr(std::vector<T> & x)
std::shared_ptr<ndarray<T,2>> new_array_ptr(std::vector<std::vector<T>> & x2)
```

Create a new N -dimensional array of type `T` and wrap it in a shared pointer.

Examples:

```
auto a1 = new_array_ptr<double,1>({ 1.1, 2.2 , 3.3, 4.4 } );
auto a2 = new_array_ptr<double,2>({ { 1.1, 2.2 }, { 3.3, 4.4 } } );
```

```
std::vector<double> x(5);
// ....
auto a = monty::new_array_ptr<double>(x);
// Now we can use a in Fusion calls, eg:
auto product = Expr::dot(a, v);
```

```
std::vector<std::vector<double>> x({{1,1,1,1,1}, {1,2,3,4,5}});
// ....
auto product = Expr::mul(monty::new_array_ptr<double>(x), v);
```

Parameters

- `init (init_t)` – An initializer list. The type `init_t` is an N -fold nested initializer list of type `std::initializer_list`. For example, for 2-dimensional arrays this is an initializer list of initializer lists, and so forth.
- `x (std::vector<T>)` – A vector of type `T`. The new array will have length `x.size()`.
- `x2 (std::vector<std::vector<T>>)` – A vector of type `T`. All rows in `x2` must be vectors of the same length, otherwise an exception will be thrown.

`monty::new_vector_from_array_ptr`

```
std::vector<T> new_vector_from_array_ptr(std::shared_ptr<ndarray<T,1>> & a) {
```

Convert a *Fusion* array into a C++ vector.

Examples:

```
auto a = std::make_shared<ndarray<double,1>>(shape(5));
// ....
std::vector<double> v = monty::new_vector_from_array_ptr(a);
```

Parameters *a* (`std::shared_ptr<ndarray<T,1>>`) – A one-dimensional array.

`monty::shape`

```
shape_t<N> shape(int i1, ..., int iN)
```

Create a new shape with the given dimensions.

Example:

```
auto s = shape(2,3,4);
```

Parameters *i1, ..., iN* (`int`) – The dimensions.

14.2 Class list

Common

- *Constraint*: Abstract base class for Constraint objects.
- *Domain*: Base class for variable and constraint domains.
- *Expr*: Represents a linear expression and provides linear operators.
- *Expression*: Abstract base class for all objects which can be used as linear expressions.
- *Matrix*: Base class for all matrix objects.
- *Model*: The object containing all data related to a single optimization model.
- *Param*: Provides static methods for manipulating parameters
- *Parameter*: Abstract class representing model parameters whose values can be modified.
- *Set*: Handles shapes.
- *Var*: Provides basic operations on variable objects.
- *Variable*: Abstract base class for Variable objects.

Infrequent

- *BaseExpression*: Base class for most expressions
- *BaseVariable*: Abstract base class for Variable objects with default implementations.
- *BoundInterfaceConstraint*: Interface to either the upper bound or the lower bound of a ranged constraint.
- *BoundInterfaceVariable*: Interface to either the upper bound or the lower bound of a ranged variable.
- *ConeDomain*: A domain representing a vector cone.
- *ConicConstraint*: Represent a conic constraint.
- *ConicVariable*: Represent a conic variable.
- *LinPSDDomain*: Represent a linear PSD domain.
- *LinearConstraint*: An object representing a block of linear constraints of the same type.
- *LinearDomain*: Represent a domain defined by linear constraints
- *LinearPSDConstraint*: Represents a semidefinite conic constraint.

- *LinearPSDVariable*: This class represents a positive semidefinite variable.
- *LinearVariable*: An object representing a block of linear variables of the same type.
- *ModelConstraint*: Represent a block of constraints.
- *ModelVariable*: Represent a block of variables.
- *NDSparseArray*: Representation of a sparse n-dimensional array.
- *PSDConstraint*: Represents a semidefinite conic constraint.
- *PSDDomain*: Represent the domain of PSD matrices.
- *PSDVariable*: This class represents a positive semidefinite variable.
- *RangeDomain*: The range domain represents a ranged subset of the euclidian space.
- *RangedConstraint*: Represents a ranged constraint.
- *RangedVariable*: Represents a ranged variable.
- *SliceConstraint*: An alias for a subset of constraints from a single ModelConstraint.
- *SliceVariable*: An alias for a subset of variables from a single model variable.
- *SymLinearVariable*: An object representing a block of linear variables of the same type.
- *SymRangedVariable*: Defines a symmetric ranged variable.
- *SymmetricLinearDomain*: Represent a linear domain with symmetry.
- *SymmetricRangeDomain*: Represent a ranged domain with symmetry.
- *SymmetricVariable*: An object representing a symmetric variable.
- *WorkStack*: Stack object used to store expression evaluations. For internal use.

14.2.1 Class BaseExpression

`mosek::fusion::BaseExpression`
Base class for most expressions

Members *BaseExpression.eval* – Evaluate the expression and push the result onto the work stack.

BaseExpression.getDim – Return the d'th dimension in the expression.

BaseExpression.getND – Return the number of dimensions in the expression.

BaseExpression.getShape – Get the shape of the expression.

BaseExpression.getSize – Return the total number of elements in the expression (the product of the dimensions).

BaseExpression.index – Get a single element in the expression.

BaseExpression.pick – Pick a number of elements from the expression.

BaseExpression.slice – Get a slice of the expression.

BaseExpression.toString – Return a string representation of the expression object.

Implemented by *Expr*

`BaseExpression.eval`

```
void eval(WorkStack::t rs, WorkStack::t ws, WorkStack::t xs)
```

Evaluate the expression and push the result onto the `rs` work stack.

Parameters

- `rs` (*WorkStack*) – The stack where the result of the evaluation is stored.

- `ws` (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- `xs` (*WorkStack*) – An auxiliary stack.

`BaseExpression.getDim`

```
int getDim(int d)
```

Return the d'th dimension in the expression.

Parameters `d` (int)

Return (int)

`BaseExpression.getND`

```
int getND()
```

Return the number of dimensions in the expression.

Return (int)

`BaseExpression.getShape`

```
shared_ptr<ndarray<int,1>> getShape()
```

Get the shape of the expression.

Return (int[])

`BaseExpression.getSize`

```
long long getSize()
```

Return the total number of elements in the expression (the product of the dimensions).

Return (long long)

`BaseExpression.index`

```
Expression::t index(int i)
Expression::t index(shared_ptr<ndarray<int,1>> indexes)
```

Get a single element in the expression.

Parameters

- `i` (int) – Index of the element to pick.
- `indexes` (int[]) – Multi-dimensional index of the element to pick.

Return (*Expression*)

`BaseExpression.pick`

```
Expression::t pick(shared_ptr<ndarray<int,1>> indexes)
Expression::t pick(shared_ptr<ndarray<int,2>> indexrows)
```

Picks a number of elements from the expression and returns them as a one-dimensional expression.

Parameters

- `indexes` (int[]) – Indexes of the elements to pick

- `indexrows (int[[[[]]])` – Indexes of the elements to pick. Each row defines a separate multi-dimensional index.

Return (*Expression*)

`BaseExpression.slice`

```
Expression::t slice(int first, int last)
Expression::t slice(shared_ptr<ndarray<int,1>> firsta, shared_ptr<ndarray<int,1>>
→ lasta)
```

Get a slice of the expression.

Parameters

- `first (int)` – Index of the first element in the slice.
- `last (int)` – Index of the last element in the slice plus one.
- `firsta (int[])` – Multi-dimensional index of the first element in the slice.
- `lasta (int[])` – Multi-dimensional index of the element after the end of the slice.

Return (*Expression*)

`BaseExpression.toString`

```
string toString()
```

Return a string representation of the expression object.

Return (string)

14.2.2 Class BaseVariable

`mosek::fusion::BaseVariable`

An abstract variable object. This class provides various default implementations of methods in *Variable*.

Members *BaseVariable.antiDiag* – Return the antidiagonal of a square variable matrix.

BaseVariable.asExpr – Create an expression corresponding to the variable object.

BaseVariable.diag – Return the diagonal of a square variable matrix.

BaseVariable.dual – Get the dual solution value of the variable.

BaseVariable.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.fromTril – Convert from a trilinear representation into a square variable.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.

BaseVariable.getSize – Get the total number of elements in the variable.

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any.

BaseVariable.makeInteger – Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.

BaseVariable.pick – Create a one-dimensional variable by picking a list of indexes from this variable.

BaseVariable.remove – Remove the variable from the model.

BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this variable
BaseVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension.
BaseVariable.toString – Create a string representation of the variable.
BaseVariable.transpose – Transpose the variable.
BaseVariable.tril – Convert from a square variable to a trilinear representation.

Implemented by *ModelVariable*, *SliceVariable*

BaseVariable.antidiag

```
Variable::t antidiag()
Variable::t antidiag(int index)
```

Return the antidiagonal of a square variable matrix.

Parameters *index* (int) – Index of the anti-diagonal

Return (*Variable*)

BaseVariable.asExpr

```
Expression::t asExpr()
```

Create an *Expression* object corresponding to $I \cdot V$, where I is the identity matrix and V is this variable.

Return (*Expression*)

BaseVariable.diag

```
Variable::t diag()
Variable::t diag(int index)
```

Return the diagonal of a square variable matrix.

Parameters *index* (int) – Index of the anti-diagonal

Return (*Variable*)

BaseVariable.dual

```
shared_ptr<ndarray<double,1>> dual()
```

Get the dual solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (double[])

BaseVariable.eval

```
void eval(WorkStack::t rs, WorkStack::t ws, WorkStack::t xs)
```

Evaluate the expression and push the result onto the *rs* work stack.

Parameters

- *rs* (*WorkStack*) – The stack where the result of the evaluation is stored.
- *ws* (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.

- `xs` (*WorkStack*) – An auxiliary stack.

`BaseVariable.fromTril`

```
Variable::t fromTril(int d)
Variable::t fromTril(int dim0, int d)
```

Convert from a trilinear representation into a square variable.

Parameters

- `d` (`int`) – Dimension of the square variable.
- `dim0` (`int`) – Index of the trilinear variable slices in a multi-dimensional representation.

Return (*Variable*)

`BaseVariable.getDim`

```
int getDim(int d)
```

Return the `d`'th dimension in the expression.

Parameters `d` (`int`)

Return (`int`)

`BaseVariable.getModel`

```
Model::t getModel()
```

Get the *Model* object that the variable belongs to.

Return (*Model*)

`BaseVariable.getND`

```
int getND()
```

Get the number of dimensions in the variable shape.

Return (`int`)

`BaseVariable.getShape`

```
shared_ptr<ndarray<int,1>> getShape()
```

Get the variable shape.

Return (`int[]`)

`BaseVariable.getSize`

```
long long getSize()
```

Get the total number of elements in the variable.

Return (`long long`)

`BaseVariable.index`

```

Variable::t index(int index)
Variable::t index(shared_ptr<ndarray<int,1>> index)
Variable::t index(int i0, int i1)
Variable::t index(int i0, int i1, int i2)

```

Return a variable slice of size 1 corresponding to a single element in the variable object..

Parameters

- `index (int)`
- `index (int[])`
- `i0 (int)` – Index in the first dimension of the element requested.
- `i1 (int)` – Index in the second dimension of the element requested.
- `i2 (int)` – Index in the second dimension of the element requested.

Return (*Variable*)

`BaseVariable.level`

```

shared_ptr<ndarray<double,1>> level()

```

Get the primal solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (`double[]`)

`BaseVariable.makeContinuous`

```

void makeContinuous()

```

Drop integrality constraints on the variable, if any.

`BaseVariable.makeInteger`

```

void makeInteger()

```

Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.

`BaseVariable.pick`

```

Variable::t pick(shared_ptr<ndarray<int,1>> idxs)
Variable::t pick(shared_ptr<ndarray<int,2>> midxs)
Variable::t pick(shared_ptr<ndarray<int,1>> i0, shared_ptr<ndarray<int,1>> i1)
Variable::t pick(shared_ptr<ndarray<int,1>> i0, shared_ptr<ndarray<int,1>> i1,
↳ shared_ptr<ndarray<int,1>> i2)

```

Create a one-dimensional variable by picking a list of indexes from this variable.

Parameters

- `idxs (int[])` – Indexes of the elements requested.
- `midxs (int[][])` – A sequence of multi-dimensional indexes of the elements requested.
- `i0 (int[])` – Index along the first dimension.
- `i1 (int[])` – Index along the second dimension.
- `i2 (int[])` – Index along the third dimension.

Return (*Variable*)

`BaseVariable.remove`

```
void remove()
```

Remove the variable from the model and remove it from any constraints where it appears. Using the variable object after this method has been called results in undefined behavior.

`BaseVariable.reshape`

```
Variable::t reshape(shared_ptr<ndarray<int,1>> shape)
Variable::t reshape(int dim0)
Variable::t reshape(int dim0, int dim1)
Variable::t reshape(int dim0, int dim1, int dim2)
```

Reshape the variable. The new shape must have the same total size as the current.

Parameters

- `shape (int[])` – The new shape.
- `dim0 (int)` – First dimension of new shape
- `dim1 (int)` – Second dimension of new shape
- `dim2 (int)` – Third dimension of new shape

Return (*Variable*)

`BaseVariable.setLevel`

```
void setLevel(shared_ptr<ndarray<double,1>> v)
```

Set values for an initial solution for this variable. Note that these values are buffered until the solver is called; they are not available through the `level()` methods.

Parameters `v (double[])` – An array of values to be assigned to the variable.

`BaseVariable.slice`

```
Variable::t slice(int first, int last)
Variable::t slice(shared_ptr<ndarray<int,1>> first, shared_ptr<ndarray<int,1>>
↪ last)
```

Create a slice variable by picking a range of indexes for each variable dimension.

Parameters

- `first (int)` – The index from which the slice begins.
- `first (int[])` – The index from which the slice begins.
- `last (int)` – The index after the last element of the slice.
- `last (int[])` – The index after the last element of the slice.

Return (*Variable*)

`BaseVariable.toString`

```
string toString()
```

Create a string representation of the variable.

Return (`string`)

`BaseVariable.transpose`

```
Variable::t transpose()
```

Return the transpose of the current variable. The variable must have at most two dimensions.

Return (*Variable*)

BaseVariable.tril

```
Variable::t tril()
Variable::t tril(int dim1, int dim2)
```

Convert from a square variable to a trilinear representation.

Parameters

- **dim1** (int) – First dimension in the current shape containing the square variables.
- **dim2** (int) – Second dimension in the current shape containing the square variables.

Return (*Variable*)

14.2.3 Class BoundInterfaceConstraint

mosek::fusion::BoundInterfaceConstraint

Interface to either the upper bound or the lower bound of a ranged constraint.

This class is never explicitly instantiated; it is created by a *RangedConstraint* to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The constraint

$$b_l \leq a^T x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the *RangedConstraint* object, but can be accessed through a *BoundInterfaceConstraint*.

Implements *SliceConstraint*

Members *BoundInterfaceConstraint.dual* – Get the dual solution values of the constraint.

BoundInterfaceConstraint.index – Get a single element from a one-dimensional constraint.

BoundInterfaceConstraint.slice – Create a slice constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.remove – Remove the constraint from the model.

Constraint.update – Update part of a constraint.

SliceConstraint.toString – Create a human readable string representation of the constraint.

BoundInterfaceConstraint.dual

```
shared_ptr<ndarray<double,1>> dual()
```

Get the dual solution value of the constraint as an array.

Return (double[])

BoundInterfaceConstraint.index

```
Constraint::t index(int idx)
Constraint::t index(shared_ptr<ndarray<int,1>> idxa)
```

Get a single element from a one-dimensional constraint.

Parameters

- `idx (int)` – The element index.
- `idxa (int[])` – Array of integer coordinates in each dimension.

Return (*Constraint*)

`BoundInterfaceConstraint.slice`

```
Constraint::t slice(int first, int last)
Constraint::t slice(shared_ptr<ndarray<int,1>> firsta, shared_ptr<ndarray<int,1>>
→ lasta)
```

Create a slice constraint.

Parameters

- `first (int)` – Index of the first element in the slice.
- `last (int)` – Index of the first element after the end of the slice.
- `firsta (int[])` – The indexes of first elements in the slice along each dimension.
- `lasta (int[])` – The indexes of first elements after the end of the slice along each dimension.

Return (*Constraint*)

14.2.4 Class `BoundInterfaceVariable`

`mosek::fusion::BoundInterfaceVariable`

Interface to either the upper bound or the lower bound of a ranged variable.

This class is never explicitly instantiated; is created by a *RangedVariable* to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The variable

$$b_l \leq x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the *RangedVariable* object, but can be accessed through a *BoundInterfaceVariable*.

Implements *SliceVariable*

Members *BaseVariable.asExpr* – Create an expression corresponding to the variable object.

BaseVariable.eval – Evaluate the expression and push the result onto the work stack.

BaseVariable.fromTril – Convert from a trilinear representation into a square variable.

BaseVariable.getDim – Return the d'th dimension in the expression.

BaseVariable.getModel – Get the *Model* object that the variable belongs to.

BaseVariable.getND – Get the number of dimensions in the variable shape.

BaseVariable.getShape – Get the variable shape.

BaseVariable.getSize – Get the total number of elements in the variable.

BaseVariable.index – Return a variable slice of size 1 corresponding to a single element in the variable object..

BaseVariable.level – Get the primal solution value of the variable.

BaseVariable.makeContinuous – Drop integrality constraints on the variable, if any.

BaseVariable.makeInteger – Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.
BaseVariable.remove – Remove the variable from the model.
BaseVariable.reshape – Reshape the variable. The new shape must have the same total size as the current.
BaseVariable.setLevel – Input solution values for this variable
BaseVariable.toString – Create a string representation of the variable.
BaseVariable.tril – Convert from a square variable to a trilinear representation.
BoundInterfaceVariable.antidiag – Return the antidiagonal of a square variable matrix.
BoundInterfaceVariable.diag – Return the diagonal of a square variable matrix.
BoundInterfaceVariable.dual – Get the dual solution value of the variable.
BoundInterfaceVariable.pick – Create a one-dimensional variable by picking a list of indexes from this variable.
BoundInterfaceVariable.slice – Create a slice variable by picking a range of indexes for each variable dimension.
BoundInterfaceVariable.transpose – Transpose the variable.

`BoundInterfaceVariable.antidiag`

```
Variable::t antidiag()
Variable::t antidiag(int index)
```

Return the antidiagonal of a square variable matrix.

Parameters `index (int)` – Index of the anti-diagonal
Return (*Variable*)

`BoundInterfaceVariable.diag`

```
Variable::t diag()
Variable::t diag(int index)
```

Return the diagonal of a square variable matrix.

Parameters `index (int)` – Index of the anti-diagonal
Return (*Variable*)

`BoundInterfaceVariable.dual`

```
shared_ptr<ndarray<double,1>> dual()
```

Get the dual solution value of the variable as an array. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (`double[]`)

`BoundInterfaceVariable.pick`

```
Variable::t pick(shared_ptr<ndarray<int,1>> idxs)
Variable::t pick(shared_ptr<ndarray<int,2>> midxs)
Variable::t pick(shared_ptr<ndarray<int,1>> i0, shared_ptr<ndarray<int,1>> i1)
Variable::t pick(shared_ptr<ndarray<int,1>> i0, shared_ptr<ndarray<int,1>> i1,
↳ shared_ptr<ndarray<int,1>> i2)
```

Create a one-dimensional variable by picking a list of indexes from this variable.

Parameters

- `idxs (int[])` – Indexes of the elements requested.
- `midxs (int[][])` – A sequence of multi-dimensional indexes of the elements requested.
- `i0 (int[])` – Index along the first dimension.
- `i1 (int[])` – Index along the second dimension.
- `i2 (int[])` – Index along the third dimension.

Return (*Variable*)

`BoundInterfaceVariable.slice`

```
Variable::t slice(int first, int last)
Variable::t slice(shared_ptr<ndarray<int,1>> firsta, shared_ptr<ndarray<int,1>>
↳lasta)
```

Create a slice variable by picking a range of indexes for each variable dimension.

Parameters

- `first (int)` – The index from which the slice begins.
- `last (int)` – The index after the last element of the slice.
- `firsta (int[])`
- `lasta (int[])`

Return (*Variable*)

`BoundInterfaceVariable.transpose`

```
Variable::t transpose()
```

Return the transpose of the current variable. The variable must have at most two dimensions.

Return (*Variable*)

14.2.5 Class ConeDomain

`mosek::fusion::ConeDomain`

A domain representing a vector cone.

Members *ConeDomain.axis* – Set the dimension along which the cones are created.
ConeDomain.axisIsSet – Returns true if the cone axis was set
ConeDomain.GetAxis – Get the dimension along which the cones are created.
ConeDomain.integral – Creates a domain of integral variables.

`ConeDomain.axis`

```
ConeDomain::t axis(int a)
```

Set the dimension along which the cones are created.

Parameters `a (int)`

Return (*ConeDomain*)

`ConeDomain.axisIsSet`

```
bool axisIsSet()
```

Returns true if the cone axis was set

Return (`bool`)

ConeDomain.GetAxis

```
int getAxis()
```

Get the dimension along which the cones are created.

Return (int)

ConeDomain.integral

```
ConeDomain::t integral()
```

Modify a given domain restricting its elements to be integral.

Return (*ConeDomain*)

14.2.6 Class ConicConstraint

mosek::fusion::ConicConstraint

This class represents a conic constraint of the form

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

where \mathcal{K} is a cone. Then class is never explicitly instantiated, but is created using *Model.constraint* by specifying a conic domain.

Note that a conic constraint in *Fusion* is always *dense* in the sense that all member constraints are created in the underlying optimization problem immediately.

Implements *ModelConstraint*

Members *ConicConstraint.toString* – Create a human readable string representation of the constraint.

Constraint.dual – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.slice – Create a slice constraint.

Constraint.update – Update part of a constraint.

ModelConstraint.remove – Remove the constraint from the model.

ConicConstraint.toString

```
string toString()
```

Create a human readable string representation of the constraint.

Return (string)

14.2.7 Class ConicVariable

`mosek::fusion::ConicVariable`

This class represents a conic variable of the form

$$x \in \mathcal{K}$$

where \mathcal{K} is a cone. Then class is never explicitly instantiated, but is created using `Model.variable` by specifying a conic domain.

Note that a conic variable in *Fusion* is always *dense* in the sense that all member variables are created in the underlying optimization problem immediately.

Implements `ModelVariable`

Members `BaseVariable.antidiag` – Return the antidiagonal of a square variable matrix.

`BaseVariable.asExpr` – Create an expression corresponding to the variable object.

`BaseVariable.diag` – Return the diagonal of a square variable matrix.

`BaseVariable.dual` – Get the dual solution value of the variable.

`BaseVariable.eval` – Evaluate the expression and push the result onto the work stack.

`BaseVariable.fromTril` – Convert from a trilinear representation into a square variable.

`BaseVariable.getDim` – Return the d'th dimension in the expression.

`BaseVariable.getModel` – Get the `Model` object that the variable belongs to.

`BaseVariable.getND` – Get the number of dimensions in the variable shape.

`BaseVariable.getShape` – Get the variable shape.

`BaseVariable.getSize` – Get the total number of elements in the variable.

`BaseVariable.index` – Return a variable slice of size 1 corresponding to a single element in the variable object..

`BaseVariable.level` – Get the primal solution value of the variable.

`BaseVariable.makeContinuous` – Drop integrality constraints on the variable, if any.

`BaseVariable.makeInteger` – Apply integrality constraints on the variable. Has no effect on elements of semidefinite matrix variables.

`BaseVariable.pick` – Create a one-dimensional variable by picking a list of indexes from this variable.

`BaseVariable.reshape` – Reshape the variable. The new shape must have the same total size as the current.

`BaseVariable.setLevel` – Input solution values for this variable

`BaseVariable.slice` – Create a slice variable by picking a range of indexes for each variable dimension.

`BaseVariable.transpose` – Transpose the variable.

`BaseVariable.tril` – Convert from a square variable to a trilinear representation.

`ConicVariable.toString` – Create a string representation of the variable.

`ModelVariable.remove` – Remove the variable from the model.

`ConicVariable.toString`

`string toString()`

Create a string representation of the variable.

Return (string)

14.2.8 Class Constraint

`mosek::fusion::Constraint`

An abstract constraint object. This is the base class for all constraint types in Fusion.

The *Constraint* object can be an interface to the normal model constraint, e.g. *LinearConstraint* and *ConicConstraint*, to slices of other constraints or to concatenations of other constraints.

Primal and dual solution values can be accessed through the *Constraint* object.

Members *Constraint.dual* – Get the dual solution values of the constraint.

Constraint.getModel – Return the model that the constraint belongs to.

Constraint.getND – Return the number of dimensions in the constraint shape.

Constraint.getShape – Return the constraint's shape.

Constraint.getSize – Return the total number of elements in the constraint.

Constraint.index – Get a single element from a constraint.

Constraint.level – Get the primal solution values of the constraint.

Constraint.remove – Remove the constraint from the model.

Constraint.slice – Create a slice constraint.

Constraint.toString – Create a human readable string representation of the constraint.

Constraint.update – Update part of a constraint.

Static members *Constraint.hstack* – Stack a number of constraints horizontally.

Constraint.stack – Stack a number of constraints.

Constraint.vstack – Stack a number of constraints vertically.

Implemented by *ModelConstraint*, *SliceConstraint*

`Constraint.dual`

```
shared_ptr<ndarray<double,1>> dual()
```

Get the dual solution values of the constraint or its slice. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (`double[]`)

`Constraint.getModel`

```
Model::t getModel()
```

Return the model that the constraint belongs to.

Return (*Model*)

`Constraint.getND`

```
int getND()
```

Return the number of dimensions in the constraint shape.

Return (`int`)

`Constraint.getShape`

```
shared_ptr<ndarray<int,1>> getShape()
```

Return the constraint's shape.

Return (`int[]`)

Constraint.getSize

```
int getSize()
```

Return the total number of elements in the constraint.

Return (int)

Constraint.hstack

```
Constraint::t Constraint::hstack(Constraint::t v1, Constraint::t v2)
Constraint::t Constraint::hstack(Constraint::t v1, Constraint::t v2, Constraint::
→t v3)
Constraint::t Constraint::hstack(shared_ptr<ndarray<Constraint::t,1>> clist)
```

Stack a number of constraints horizontally.

Parameters

- v1 (*Constraint*) – The first constraint in the stack.
- v2 (*Constraint*) – The second constraint in the stack.
- v3 (*Constraint*) – The third constraint in the stack.
- clist (*Constraint*[]) – The constraints in the stack.

Return (*Constraint*)

Constraint.index

```
Constraint::t index(int idx)
Constraint::t index(shared_ptr<ndarray<int,1>> idxa)
```

Get a single element from a one-dimensional constraint.

Parameters

- idx (int) – The index of the element.
- idxa (int[]) – A multi-dimensional index of the element.

Return (*Constraint*)

Constraint.level

```
shared_ptr<ndarray<double,1>> level()
```

Get the primal solution values of the constraint. This amounts to evaluating the Ax part of the constraint expression for the relevant solution value. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

Return (double[])

Constraint.remove

```
void remove()
```

Remove the constraint from the model. Using the constraint object after this method has been called results in undefined behavior.

Constraint.slice

```
Constraint::t slice(int first, int last)
Constraint::t slice(shared_ptr<ndarray<int,1>> firsta, shared_ptr<ndarray<int,1>>
→ lasta)
```

Create a slice constraint.

Parameters

- **first** (int) – Index of the first element in the slice.
- **last** (int) – Index of the first element after the end of the slice.
- **firsta** (int[]) – The indexes of first elements in the slice along each dimension.
- **lasta** (int[]) – The indexes of first elements after the end of the slice along each dimension.

Return (*Constraint*)

`Constraint.stack`

```
Constraint::t Constraint::stack(Constraint::t v1, Constraint::t v2, int dim)
Constraint::t Constraint::stack(Constraint::t v1, Constraint::t v2, Constraint::
→ t v3, int dim)
Constraint::t Constraint::stack(shared_ptr<ndarray<Constraint::t,1>> clist, int
→ dim)
```

Stack a number of constraints.

Parameters

- **v1** (*Constraint*) – The first constraint in the stack.
- **v2** (*Constraint*) – The second constraint in the stack.
- **dim** (int) – The dimension in which to stack, 0 means vertically.
- **v3** (*Constraint*) – The third constraint in the stack.
- **clist** (*Constraint*[]) – The constraints in the stack.

Return (*Constraint*)

`Constraint.toString`

```
string toString()
```

Create a human readable string representation of the constraint.

Return (string)

`Constraint.update`

```
void update(Expression::t expr, Variable::t x)
void update(Expression::t expr, Variable::t x, bool bfixupdate)
void update(Expression::t expr)
void update(shared_ptr<ndarray<double,1>> bfix)
```

Update entire or part of the left-hand side of a constraint (the expression). See [Sec. 7.10](#) for a tutorial.

If only **expr** is given, replace the entire previous expression with **expr**. The shape of **expr** must match the shape of the constraint.

If only **bfix** is given, replace the constant term of the expression with **bfix**. The length of **bfix** must match the size of the expression.

If **x** is given, replace all columns in the constraint defined by **x** by terms defined by **expr**, possibly including constant terms. Currently it is only possible to update linear columns. Attempting to update columns of semidefinite variables will result in an error.

Set the dimension along which the cones are created. If this conic domain is used for a variable or expression of dimension d , then the conic constraint will be applicable to all vectors obtained by fixing the coordinates other than a -th and moving along the a -th coordinate. If $d = 2$ this can be used to define the conditions “every row of the matrix is in a cone” and “every column of a matrix is in a cone”.

The default is the last dimension $a = d - 1$.

Parameters

- **c** (*ConeDomain*) – A conic domain.
- **a** (int) – The axis.

Return (*ConeDomain*)

Domain.binary

```
RangeDomain::t Domain::binary(int n)
RangeDomain::t Domain::binary(int m, int n)
RangeDomain::t Domain::binary(shared_ptr<ndarray<int,1>> dims)
RangeDomain::t Domain::binary()
```

Create a domain of binary variables. A binary domain can only be used when creating variables, but is not allowed in a constraint. Another way of restricting variables to be integers is the method *Variable.makeInteger*.

Parameters

- **n** (int) – Dimension size.
- **m** (int) – Dimension size.
- **dims** (int[]) – A list of dimension sizes.

Return (*RangeDomain*)

Domain.equalsTo

```
LinearDomain::t Domain::equalsTo(double b)
LinearDomain::t Domain::equalsTo(double b, int n)
LinearDomain::t Domain::equalsTo(double b, int m, int n)
LinearDomain::t Domain::equalsTo(double b, shared_ptr<ndarray<int,1>> dims)
LinearDomain::t Domain::equalsTo(shared_ptr<ndarray<double,1>> a1)
LinearDomain::t Domain::equalsTo(shared_ptr<ndarray<double,2>> a2)
LinearDomain::t Domain::equalsTo(shared_ptr<ndarray<double,1>> a1, shared_ptr
→<ndarray<int,1>> dims)
LinearDomain::t Domain::equalsTo(Matrix::t mx)
```

Defines the domain consisting of a fixed point.

Parameters

- **b** (double) – A single value. This is scalable: it means that each element in the variable or constraint is fixed to b .
- **n** (int) – Dimension size.
- **m** (int) – Dimension size.
- **dims** (int[]) – A list of dimension sizes.
- **a1** (double[]) – A one-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- **a2** (double[][]) – A two-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- **mx** (*Matrix*) – A matrix of bound values. The shape must match the variable or constraint with which it is used.

Return (*LinearDomain*)

Domain.greaterThan

```
LinearDomain::t Domain::greaterThan(double b)
LinearDomain::t Domain::greaterThan(double b, int n)
LinearDomain::t Domain::greaterThan(double b, int m, int n)
LinearDomain::t Domain::greaterThan(double b, shared_ptr<ndarray<int,1>> dims)
LinearDomain::t Domain::greaterThan(shared_ptr<ndarray<double,1>> a1)
LinearDomain::t Domain::greaterThan(shared_ptr<ndarray<double,2>> a2)
LinearDomain::t Domain::greaterThan(shared_ptr<ndarray<double,1>> a1, shared_ptr
→<ndarray<int,1>> dims)
LinearDomain::t Domain::greaterThan(Matrix::t mx)
```

Defines the domain specified by a lower bound in each dimension.

Parameters

- **b** (double) – A single value. This is scalable: it means that each element in the variable or constraint is greater than or equal to b .
- **n** (int) – Dimension size.
- **m** (int) – Dimension size.
- **dims** (int[]) – A list of dimension sizes.
- **a1** (double[]) – A one-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- **a2** (double[][]) – A two-dimensional array of bounds. The shape must match the variable or constraint with which it is used.
- **mx** (*Matrix*) – A matrix of bound values. The shape must match the variable or constraint with which it is used.

Return (*LinearDomain*)

Domain.inExpCone

```
ConeDomain::t Domain::inExpCone()
ConeDomain::t Domain::inExpCone(int m)
ConeDomain::t Domain::inExpCone(shared_ptr<ndarray<int,1>> dims)
```

Defines the domain of dual exponential cones:

$$\left\{ x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} e^{x_2/x_3}, x_1 > 0, x_3 < 0 \right\}$$

The conic domain scales as follows. If a variable or expression constrained to an exponential cone is not a single vector but a d -dimensional variable then the conic domain is applicable to all vectors obtained by fixing the first $d - 1$ coordinates and moving along the last coordinate. If $d = 2$ it means that each row of a matrix must belong to a cone. See also *Domain.axis*.

If m was given the domain is a product of m such cones.

Parameters

- **m** (int) – The number of cones (default 1).
- **dims** (int[]) – Shape of the domain.

Return (*ConeDomain*)

Domain.inDPowerCone

```
ConeDomain::t Domain::inDPowerCone(double alpha)
ConeDomain::t Domain::inDPowerCone(double alpha, int m)
ConeDomain::t Domain::inDPowerCone(double alpha, shared_ptr<ndarray<int,1>> dims)
```


Return (*Expression*)

Expr.condense

```
Expression::t Expr::condense(Expression::t e)
```

Flatten expression and remove all structural zeros.

Parameters *e* (*Expression*) – Expression to be condensed.

Return (*Expression*)

Expr.constTerm

```
Expression::t Expr::constTerm(shared_ptr<ndarray<double,1>> vals1)
Expression::t Expr::constTerm(shared_ptr<ndarray<double,2>> vals2)
Expression::t Expr::constTerm(int size, double val)
Expression::t Expr::constTerm(shared_ptr<ndarray<int,1>> shp, double val)
Expression::t Expr::constTerm(shared_ptr<ndarray<int,1>> shp, shared_ptr<ndarray
→<int,2>> sparsity, shared_ptr<ndarray<double,1>> vals1)
Expression::t Expr::constTerm(shared_ptr<ndarray<int,1>> shp, shared_ptr<ndarray
→<int,2>> sparsity, double val)
Expression::t Expr::constTerm(double val)
Expression::t Expr::constTerm(Matrix::t m)
Expression::t Expr::constTerm(NDSparseArray::t nda)
```

Create an expression consisting of a constant vector of values.

Parameters

- *vals1* (double[]) – A vector initializing the expression.
- *vals2* (double[][]) – An array initializing the expression.
- *size* (int) – Length of the vector to be constructed.
- *val* (double) – A scalar value to be repeated in all entries of the expression.
- *shp* (int[]) – Defines the shape of the expression.
- *sparsity* (int[][]) – Sparsity pattern.
- *m* (*Matrix*) – A matrix of values initializing the expression.
- *nda* (*NDSparseArray*) – An multi-dimensional sparse array initializing the expression.

Return (*Expression*)

Expr.dot

```
Expression::t Expr::dot(Parameter::t p, Expression::t e)
Expression::t Expr::dot(Expression::t e, Parameter::t p)
Expression::t Expr::dot(shared_ptr<ndarray<double,1>> c1, Expression::t e)
Expression::t Expr::dot(shared_ptr<ndarray<double,2>> c2, Expression::t e)
Expression::t Expr::dot(NDSparseArray::t nda, Expression::t e)
Expression::t Expr::dot(Matrix::t m, Expression::t e)
Expression::t Expr::dot(Expression::t e, shared_ptr<ndarray<double,1>> c1)
Expression::t Expr::dot(Expression::t e, NDSparseArray::t nda)
Expression::t Expr::dot(Expression::t e, shared_ptr<ndarray<double,2>> c2)
Expression::t Expr::dot(Expression::t e, Matrix::t m)
```

Return an object representing the inner product (dot product) $x^T y = \sum_{i=1}^n x_i y_i$ of two objects x, y of size n .

Both arguments must have the same length when flattened. In particular, they can be two vectors of the same length or two matrices of the same shape.

Parameters

- *p* (*Parameter*) – A parameter.
- *e* (*Expression*) – An expression object.
- *c1* (`double[]`) – A one-dimensional coefficient vector.
- *c2* (`double[][]`) – A two-dimensional coefficient array.
- *nda* (*NDSparseArray*) – A multi-dimensional sparse array.
- *m* (*Matrix*) – A matrix object.

Return (*Expression*)

`Expr.eval`

```
void eval(WorkStack::t rs, WorkStack::t ws, WorkStack::t xs)
```

Evaluate the expression and push the result onto the `rs` work stack.

Parameters

- *rs* (*WorkStack*) – The stack where the result of the evaluation is stored.
- *ws* (*WorkStack*) – The stack used by evaluation to perform intermediate computations. It will be returned in the same state as when the function is called.
- *xs* (*WorkStack*) – An auxiliary stack.

`Expr.flatten`

```
Expression::t Expr::flatten(Expression::t e)
```

Reshape the expression into a vector.

Parameters *e* (*Expression*) – The expression to be flattened.

Return (*Expression*)

`Expr.getND`

```
int getND()
```

Return the number of dimensions in the expression.

Return (`int`)

`Expr.getShape`

```
shared_ptr<ndarray<int,1>> getShape()
```

Return the shape of the expression

Return (`int[]`)

`Expr.hstack`

```
Expression::t Expr::hstack(shared_ptr<ndarray<Expression::t,1>> exprs)
Expression::t Expr::hstack(Expression::t e1, Expression::t e2)
Expression::t Expr::hstack(Expression::t e1, double a2)
Expression::t Expr::hstack(double a1, Expression::t e2)
Expression::t Expr::hstack(double a1, double a2, Expression::t e3)
Expression::t Expr::hstack(double a1, Expression::t e2, double a3)
Expression::t Expr::hstack(double a1, Expression::t e2, Expression::t e3)
```

(continues on next page)

Example `M->setSolverParam("presolveMaxNumPass", -1)`
Generic name `MSK_IPAR_PRESOLVE_MAX_NUM_PASS`
Groups *Presolve*

"presolveUse"

Controls whether the presolve is applied to a problem before it is optimized.

Default *"free"*
Accepted *"off", "on", "free"*
Example `M->setSolverParam("presolveUse", "free")`
Generic name `MSK_IPAR_PRESOLVE_USE`
Groups *Overall solver, Presolve*

"removeUnusedSolutions"

Removes unused solutions before the optimization is performed.

Default *"off"*
Accepted *"on", "off"*
Example `M->setSolverParam("removeUnusedSolutions", "off")`
Generic name `MSK_IPAR_REMOVE_UNUSED_SOLUTIONS`
Groups *Overall system*

"simBasisFactorUse"

Controls whether an LU factorization of the basis is used in a hot-start. Forcing a refactorization sometimes improves the stability of the simplex optimizers, but in most cases there is a performance penalty.

Default *"on"*
Accepted *"on", "off"*
Example `M->setSolverParam("simBasisFactorUse", "on")`
Generic name `MSK_IPAR_SIM_BASIS_FACTOR_USE`
Groups *Simplex optimizer*

"simDegen"

Controls how aggressively degeneration is handled.

Default *"free"*
Accepted *"none", "free", "aggressive", "moderate", "minimum"*
Example `M->setSolverParam("simDegen", "free")`
Generic name `MSK_IPAR_SIM_DEGEN`
Groups *Simplex optimizer*

"simDualCrash"

Controls whether crashing is performed in the dual simplex optimizer. If this parameter is set to x , then a crash will be performed if a basis consists of more than $(100 - x) \bmod f_v$ entries, where f_v is the number of fixed variables.

Default 90
Accepted `[0; +inf]`
Example `M->setSolverParam("simDualCrash", 90)`
Generic name `MSK_IPAR_SIM_DUAL_CRASH`
Groups *Dual simplex*

"simDualPhaseoneMethod"

An experimental feature.

Default 0
Accepted `[0; 10]`
Example `M->setSolverParam("simDualPhaseoneMethod", 0)`

Default 10000000
Accepted [0; +inf]
Example M->setSolverParam("simMaxIterations", 10000000)
Generic name MSK_IPAR_SIM_MAX_ITERATIONS
Groups *Simplex optimizer, Termination criteria*

"simMaxNumSetbacks"

Controls how many set-backs are allowed within a simplex optimizer. A set-back is an event where the optimizer moves in the wrong direction. This is impossible in theory but may happen due to numerical problems.

Default 250
Accepted [0; +inf]
Example M->setSolverParam("simMaxNumSetbacks", 250)
Generic name MSK_IPAR_SIM_MAX_NUM_SETBACKS
Groups *Simplex optimizer*

"simNonSingular"

Controls if the simplex optimizer ensures a non-singular basis, if possible.

Default *"on"*
Accepted *"on", "off"*
Example M->setSolverParam("simNonSingular", "on")
Generic name MSK_IPAR_SIM_NON_SINGULAR
Groups *Simplex optimizer*

"simPrimalCrash"

Controls whether crashing is performed in the primal simplex optimizer. In general, if a basis consists of more than (100-this parameter value)% fixed variables, then a crash will be performed.

Default 90
Accepted [0; +inf]
Example M->setSolverParam("simPrimalCrash", 90)
Generic name MSK_IPAR_SIM_PRIMAL_CRASH
Groups *Primal simplex*

"simPrimalPhaseoneMethod"

An experimental feature.

Default 0
Accepted [0; 10]
Example M->setSolverParam("simPrimalPhaseoneMethod", 0)
Generic name MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD
Groups *Simplex optimizer*

"simPrimalRestrictSelection"

The primal simplex optimizer can use a so-called restricted selection/pricing strategy to choose the outgoing variable. Hence, if restricted selection is applied, then the primal simplex optimizer first choose a subset of all the potential incoming variables. Next, for some time it will choose the incoming variable only among the subset. From time to time the subset is redefined. A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Default 50
Accepted [0; 100]
Example M->setSolverParam("simPrimalRestrictSelection", 50)
Generic name MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION
Groups *Primal simplex*

PrimalInfeasibleOrUnbounded

The problem is primal infeasible or unbounded.

SolutionStatus

Defines properties of either a primal or a dual solution. A model may contain multiple solutions which may have different status. Specifically, there will be individual solutions, and thus solution statuses, for the interior-point, simplex and integer solvers.

Undefined

Undefined solution. This means that no values exist for the relevant solution.

Unknown

The solution status is unknown; this will happen if the user inputs values or a solution is read from a file or the solver stalled.

Optimal

The solution values are feasible and optimal.

Feasible

The solution is feasible.

Certificate

The solution is a certificate of infeasibility.

IllposedCert

The solution is a certificate of illposedness.

SolutionType

Used when requesting a specific solution from a *Model*.

Default

Auto-select the default solution; usually this will be the integer solution, if available, otherwise the basic solution, if available, otherwise the interior-point solution.

Basic

Select the basic solution.

Interior

Select the interior-point solution.

Integer

Select the integer solution.

14.6 Constants

14.6.1 Basis identification

"never"

Never do basis identification.

"always"

Basis identification is always performed even if the interior-point optimizer terminates abnormally.

"noError"

Basis identification is performed if the interior-point optimizer terminates without an error.

"ifFeasible"

Basis identification is not performed if the interior-point optimizer terminates with a problem status saying that the problem is primal or dual infeasible.

"reserved"

Not currently in use.

14.6.2 Bound keys

"lo"	The constraint or variable has a finite lower bound and an infinite upper bound.
"up"	The constraint or variable has an infinite lower bound and an finite upper bound.
"fx"	The constraint or variable is fixed.
"fr"	The constraint or variable is free.
"ra"	The constraint or variable is ranged.

14.6.3 Mark

"lo"	The lower bound is selected for sensitivity analysis.
"up"	The upper bound is selected for sensitivity analysis.

14.6.4 Degeneracy strategies

"none"	The simplex optimizer should use no degeneration strategy.
"free"	The simplex optimizer chooses the degeneration strategy.
"aggressive"	The simplex optimizer should use an aggressive degeneration strategy.
"moderate"	The simplex optimizer should use a moderate degeneration strategy.
"minimum"	The simplex optimizer should use a minimum degeneration strategy.

14.6.5 Transposed matrix.

"no"	No transpose is applied.
"yes"	A transpose is applied.

14.6.6 Triangular part of a symmetric matrix.

"lo"	Lower part.
"up"	Upper part.

14.6.7 Problem reformulation.

"on"	Allow the simplex optimizer to reformulate the problem.
"off"	Disallow the simplex optimizer to reformulate the problem.
"free"	The simplex optimizer can choose freely.
"aggressive"	The simplex optimizer should use an aggressive reformulation strategy.

"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.

"simObj"
Objective value reported by the simplex optimizer.

"simPrimalTime"
Time spent in the primal simplex optimizer since invoking it.

"simTime"
Time spent in the simplex optimizer since invoking it.

"solBasDualObj"
Dual objective value of the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasDviolcon"
Maximal dual bound violation for x^c in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasDviolvar"
Maximal dual bound violation for x^x in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasNrmBarx"
Infinity norm of \overline{X} in the basic solution.

"solBasNrmSlc"
Infinity norm of s_l^c in the basic solution.

"solBasNrmSlx"
Infinity norm of s_l^x in the basic solution.

"solBasNrmSuc"
Infinity norm of s_u^c in the basic solution.

"solBasNrmSux"
Infinity norm of s_u^X in the basic solution.

"solBasNrmXc"
Infinity norm of x^c in the basic solution.

"solBasNrmXx"
Infinity norm of x^x in the basic solution.

"solBasNrmY"
Infinity norm of y in the basic solution.

"solBasPrimalObj"
Primal objective value of the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasPviolcon"
Maximal primal bound violation for x^c in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solBasPviolvar"
Maximal primal bound violation for x^x in the basic solution. Updated if *autoUpdateSolInfo* is set .

"solItgNrmBarx"
Infinity norm of \overline{X} in the integer solution.

"solItgNrmXc"
Infinity norm of x^c in the integer solution.

"solItgNrmXx"
Infinity norm of x^x in the integer solution.

"solItgPrimalObj"
Primal objective value of the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolbarvar"
Maximal primal bound violation for \overline{X} in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolcon"
Maximal primal bound violation for x^c in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolcones"
Maximal primal violation for primal conic constraints in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolitg"
Maximal violation for the integer constraints in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItgPviolvar"
 Maximal primal bound violation for x^x in the integer solution. Updated if *autoUpdateSolInfo* is set .

"solItrDualObj"
 Dual objective value of the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolbarvar"
 Maximal dual bound violation for \bar{X} in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolcon"
 Maximal dual bound violation for x^c in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolcones"
 Maximal dual violation for dual conic constraints in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrDviolvar"
 Maximal dual bound violation for x^x in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrNrmBars"
 Infinity norm of \bar{S} in the interior-point solution.

"solItrNrmBarx"
 Infinity norm of \bar{X} in the interior-point solution.

"solItrNrmSlc"
 Infinity norm of s_l^c in the interior-point solution.

"solItrNrmSlx"
 Infinity norm of s_l^x in the interior-point solution.

"solItrNrmSnx"
 Infinity norm of s_n^x in the interior-point solution.

"solItrNrmSuc"
 Infinity norm of s_u^c in the interior-point solution.

"solItrNrmSux"
 Infinity norm of s_u^X in the interior-point solution.

"solItrNrmXc"
 Infinity norm of x^c in the interior-point solution.

"solItrNrmXx"
 Infinity norm of x^x in the interior-point solution.

"solItrNrmY"
 Infinity norm of y in the interior-point solution.

"solItrPrimalObj"
 Primal objective value of the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolbarvar"
 Maximal primal bound violation for \bar{X} in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolcon"
 Maximal primal bound violation for x^c in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolcones"
 Maximal primal violation for primal conic constraints in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"solItrPviolvar"
 Maximal primal bound violation for x^x in the interior-point solution. Updated if *autoUpdateSolInfo* is set .

"toConicTime"
 Time spent in the last to conic reformulation.

14.6.21 License feature

"pts"
Base system.
"pton"
Conic extension.

14.6.22 Long integer information items.

"biCleanDualDegIter"
Number of dual degenerate clean iterations performed in the basis identification.
"biCleanDualIter"
Number of dual clean iterations performed in the basis identification.
"biCleanPrimalDegIter"
Number of primal degenerate clean iterations performed in the basis identification.
"biCleanPrimalIter"
Number of primal clean iterations performed in the basis identification.
"biDualIter"
Number of dual pivots performed in the basis identification.
"biPrimalIter"
Number of primal pivots performed in the basis identification.
"intpntFactorNumNz"
Number of non-zeros in factorization.
"mioAnz"
Number of non-zero entries in the constraint matrix of the problem to be solved by the mixed-integer optimizer.
"mioIntpntIter"
Number of interior-point iterations performed by the mixed-integer optimizer.
"mioPresolvedAnz"
Number of non-zero entries in the constraint matrix of the problem after the mixed-integer optimizer's presolve.
"mioSimplexIter"
Number of simplex iterations performed by the mixed-integer optimizer.
"rdNumanz"
Number of non-zeros in A that is read.
"rdNumqnz"
Number of Q non-zeros.

14.6.23 Integer information items.

"anaProNumCon"
Number of constraints in the problem.
"anaProNumConEq"
Number of equality constraints.
"anaProNumConFr"
Number of unbounded constraints.
"anaProNumConLo"
Number of constraints with a lower bound and an infinite upper bound.
"anaProNumConRa"
Number of constraints with finite lower and upper bounds.
"anaProNumConUp"
Number of constraints with an upper bound and an infinite lower bound.
"anaProNumVar"
Number of variables in the problem.
"anaProNumVarBin"
Number of binary (0-1) variables.
"anaProNumVarCont"
Number of continuous variables.

"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).


```

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

```


15.7.1 *jtask* format

It stores a problem instance. The *jtask* format contains the same information as a *task format*. Even though a *jtask* file is human-readable, we do not recommend users to create it by hand, but to rely on MOSEK.

15.7.2 *jsol* format

It stores a problem solution. The *jsol* format contains all solutions and information items.

15.7.3 A *jtask* example

In Listing 15.6 we present a file in the *jtask* format that corresponds to the sample problem from `lo1.lp`. The listing has been formatted for readability.

Listing 15.6: A formatted *jtask* file for the `lo1.lp` example.

```
{
  "$schema": "http://mosek.com/json/schema#",
  "Task/INFO": {
    "taskname": "lo1",
    "numvar": 4,
    "numcon": 3,
    "numcone": 0,
    "numbarvar": 0,
    "numanz": 9,
    "numsymmat": 0,
    "mosekver": [
      8,
      0,
      0,
      9
    ]
  },
  "Task/data": {
    "var": {
      "name": [
        "x1",
        "x2",
        "x3",
        "x4"
      ],
      "bk": [
        "lo",
        "ra",
        "lo",
        "lo"
      ],
      "b1": [
        0.0,
        0.0,
        0.0,
        0.0
      ],
      "bu": [
        1e+30,
        1e+1,
        1e+30,

```

(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,

```

(continues on next page)

```

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

```

```

"INTPNT_FACTOR_DEBUG_LVL":0,
"INTPNT_FACTOR_METHOD":0,
"INTPNT_HOTSTART":"INTPNT_HOTSTART_NONE",
"INTPNT_MAX_ITERATIONS":400,
"INTPNT_MAX_NUM_COR":-1,
"INTPNT_MAX_NUM_REFINEMENT_STEPS":-1,
"INTPNT_OFF_COL_TRH":40,
"INTPNT_ORDER_METHOD":"ORDER_METHOD_FREE",
"INTPNT_REGULARIZATION_USE":"ON",
"INTPNT_SCALING":"SCALING_FREE",
"INTPNT_SOLVE_FORM":"SOLVE_FREE",
"INTPNT_STARTING_POINT":"STARTING_POINT_FREE",
"LIC_TRH_EXPIRY_WRN":7,
"LICENSE_DEBUG":"OFF",
"LICENSE_PAUSE_TIME":0,
"LICENSE_SUPPRESS_EXPIRE_WRNS":"OFF",
"LICENSE_WAIT":"OFF",
"LOG":10,
"LOG_ANA_PRO":1,
"LOG_BI":4,
"LOG_BI_FREQ":2500,
"LOG_CHECK_CONVEXITY":0,
"LOG_CONCURRENT":1,
"LOG_CUT_SECOND_OPT":1,
"LOG_EXPAND":0,
"LOG_FACTOR":1,
"LOG_FEAS_REPAIR":1,
"LOG_FILE":1,
"LOG_HEAD":1,
"LOG_INFEAS_ANA":1,
"LOG_INTPNT":4,
"LOG_MIO":4,
"LOG_MIO_FREQ":1000,
"LOG_OPTIMIZER":1,
"LOG_ORDER":1,
"LOG_PRESOLVE":1,
"LOG_RESPONSE":0,
"LOG_SENSITIVITY":1,
"LOG_SENSITIVITY_OPT":0,
"LOG_SIM":4,
"LOG_SIM_FREQ":1000,
"LOG_SIM_MINOR":1,
"LOG_STORAGE":1,
"MAX_NUM_WARNINGS":10,
"MIO_BRANCH_DIR":"BRANCH_DIR_FREE",
"MIO_CONSTRUCT_SOL":"OFF",
"MIO_CUT_CLIQUE":"ON",
"MIO_CUT_CMIR":"ON",
"MIO_CUT_GMI":"ON",
"MIO_CUT_KNAPSACK_COVER":"OFF",
"MIO_HEURISTIC_LEVEL":-1,
"MIO_MAX_NUM_BRANCHES":-1,
"MIO_MAX_NUM_RELAXS":-1,
"MIO_MAX_NUM_SOLUTIONS":-1,
"MIO_MODE":"MIO_MODE_SATISFIED",
"MIO_MT_USER_CB":"ON",

```

```

"MIO_NODE_OPTIMIZER": "OPTIMIZER_FREE",
"MIO_NODE_SELECTION": "MIO_NODE_SELECTION_FREE",
"MIO_PERSPECTIVE_REFORMULATE": "ON",
"MIO_PROBING_LEVEL": -1,
"MIO_RINS_MAX_NODES": -1,
"MIO_ROOT_OPTIMIZER": "OPTIMIZER_FREE",
"MIO_ROOT_REPEAT_PRESOLVE_LEVEL": -1,
"MT_SPINCOUNT": 0,
"NUM_THREADS": 0,
"OPF_MAX_TERMS_PER_LINE": 5,
"OPF_WRITE_HEADER": "ON",
"OPF_WRITE_HINTS": "ON",
"OPF_WRITE_PARAMETERS": "OFF",
"OPF_WRITE_PROBLEM": "ON",
"OPF_WRITE_SOL_BAS": "ON",
"OPF_WRITE_SOL_ITG": "ON",
"OPF_WRITE_SOL_ITR": "ON",
"OPF_WRITE_SOLUTIONS": "OFF",
"OPTIMIZER": "OPTIMIZER_FREE",
"PARAM_READ_CASE_NAME": "ON",
"PARAM_READ_IGN_ERROR": "OFF",
"PRESOLVE_ELIMINATOR_MAX_FILL": -1,
"PRESOLVE_ELIMINATOR_MAX_NUM_TRIES": -1,
"PRESOLVE_LEVEL": -1,
"PRESOLVE_LINDEP_ABS_WORK_TRH": 100,
"PRESOLVE_LINDEP_REL_WORK_TRH": 100,
"PRESOLVE_LINDEP_USE": "ON",
"PRESOLVE_MAX_NUM_REDUCTIONS": -1,
"PRESOLVE_USE": "PRESOLVE_MODE_FREE",
"PRIMAL_REPAIR_OPTIMIZER": "OPTIMIZER_FREE",
"QO_SEPARABLE_REFORMULATION": "OFF",
"READ_DATA_COMPRESSED": "COMPRESS_FREE",
"READ_DATA_FORMAT": "DATA_FORMAT_EXTENSION",
"READ_DEBUG": "OFF",
"READ_KEEP_FREE_CON": "OFF",
"READ_LP_DROP_NEW_VARS_IN_BOU": "OFF",
"READ_LP_QUOTED_NAMES": "ON",
"READ_MPS_FORMAT": "MPS_FORMAT_FREE",
"READ_MPS_WIDTH": 1024,
"READ_TASK_IGNORE_PARAM": "OFF",
"SENSITIVITY_ALL": "OFF",
"SENSITIVITY_OPTIMIZER": "OPTIMIZER_FREE_SIMPLEX",
"SENSITIVITY_TYPE": "SENSITIVITY_TYPE_BASIS",
"SIM_BASIS_FACTOR_USE": "ON",
"SIM_DEGEN": "SIM_DEGEN_FREE",
"SIM_DUAL_CRASH": 90,
"SIM_DUAL_PHASEONE_METHOD": 0,
"SIM_DUAL_RESTRICT_SELECTION": 50,
"SIM_DUAL_SELECTION": "SIM_SELECTION_FREE",
"SIM_EXPLOIT_DUPVEC": "SIM_EXPLOIT_DUPVEC_OFF",
"SIM_HOTSTART": "SIM_HOTSTART_FREE",
"SIM_HOTSTART_LU": "ON",
"SIM_INTEGER": 0,
"SIM_MAX_ITERATIONS": 10000000,
"SIM_MAX_NUM_SETBACKS": 250,
"SIM_NON_SINGULAR": "ON",

```

```

"SIM_PRIMAL_CRASH":90,
"SIM_PRIMAL_PHASEONE_METHOD":0,
"SIM_PRIMAL_RESTRICT_SELECTION":50,
"SIM_PRIMAL_SELECTION":"SIM_SELECTION_FREE",
"SIM_REFACTOR_FREQ":0,
"SIM_REFORMULATION":"SIM_REFORMULATION_OFF",
"SIM_SAVE_LU":"OFF",
"SIM_SCALING":"SCALING_FREE",
"SIM_SCALING_METHOD":"SCALING_METHOD_POW2",
"SIM_SOLVE_FORM":"SOLVE_FREE",
"SIM_STABILITY_PRIORITY":50,
"SIM_SWITCH_OPTIMIZER":"OFF",
"SOL_FILTER_KEEP_BASIC":"OFF",
"SOL_FILTER_KEEP_RANGED":"OFF",
"SOL_READ_NAME_WIDTH":-1,
"SOL_READ_WIDTH":1024,
"SOLUTION_CALLBACK":"OFF",
"TIMING_LEVEL":1,
"WRITE_BAS_CONSTRAINTS":"ON",
"WRITE_BAS_HEAD":"ON",
"WRITE_BAS_VARIABLES":"ON",
"WRITE_DATA_COMPRESSED":0,
"WRITE_DATA_FORMAT":"DATA_FORMAT_EXTENSION",
"WRITE_DATA_PARAM":"OFF",
"WRITE_FREE_CON":"OFF",
"WRITE_GENERIC_NAMES":"OFF",
"WRITE_GENERIC_NAMES_IO":1,
"WRITE_IGNORE_INCOMPATIBLE_CONIC_ITEMS":"OFF",
"WRITE_IGNORE_INCOMPATIBLE_ITEMS":"OFF",
"WRITE_IGNORE_INCOMPATIBLE_NL_ITEMS":"OFF",
"WRITE_IGNORE_INCOMPATIBLE_PSD_ITEMS":"OFF",
"WRITE_INT_CONSTRAINTS":"ON",
"WRITE_INT_HEAD":"ON",
"WRITE_INT_VARIABLES":"ON",
"WRITE_LP_FULL_OBJ":"ON",
"WRITE_LP_LINE_WIDTH":80,
"WRITE_LP_QUOTED_NAMES":"ON",
"WRITE_LP_STRICT_FORMAT":"OFF",
"WRITE_LP_TERMS_PER_LINE":10,
"WRITE_MPS_FORMAT":"MPS_FORMAT_FREE",
"WRITE_MPS_INT":"ON",
"WRITE_PRECISION":15,
"WRITE_SOL_BARVARIABLES":"ON",
"WRITE_SOL_CONSTRAINTS":"ON",
"WRITE_SOL_HEAD":"ON",
"WRITE_SOL_IGNORE_INVALID_NAMES":"OFF",
"WRITE_SOL_VARIABLES":"ON",
"WRITE_TASK_INC_SOL":"ON",
"WRITE_XML_MODE":"WRITE_XML_MODE_ROW"
},
"dparam":{
  "ANA_SOL_INFEAS_TOL":1e-6,
  "BASIS_REL_TOL_S":1e-12,
  "BASIS_TOL_S":1e-6,
  "BASIS_TOL_X":1e-6,
  "CHECK_CONVEXITY_REL_TOL":1e-10,

```

```

"DATA_TOL_AIJ":1e-12,
"DATA_TOL_AIJ_HUGE":1e+20,
"DATA_TOL_AIJ_LARGE":1e+10,
"DATA_TOL_BOUND_INF":1e+16,
"DATA_TOL_BOUND_WRN":1e+8,
"DATA_TOL_C_HUGE":1e+16,
"DATA_TOL_CJ_LARGE":1e+8,
"DATA_TOL_QIJ":1e-16,
"DATA_TOL_X":1e-8,
"FEASREPAIR_TOL":1e-10,
"INTPNT_CO_TOL_DFEAS":1e-8,
"INTPNT_CO_TOL_INFEAS":1e-10,
"INTPNT_CO_TOL_MU_RED":1e-8,
"INTPNT_CO_TOL_NEAR_REL":1e+3,
"INTPNT_CO_TOL_PFEAS":1e-8,
"INTPNT_CO_TOL_REL_GAP":1e-7,
"INTPNT_NL_MERIT_BAL":1e-4,
"INTPNT_NL_TOL_DFEAS":1e-8,
"INTPNT_NL_TOL_MU_RED":1e-12,
"INTPNT_NL_TOL_NEAR_REL":1e+3,
"INTPNT_NL_TOL_PFEAS":1e-8,
"INTPNT_NL_TOL_REL_GAP":1e-6,
"INTPNT_NL_TOL_REL_STEP":9.95e-1,
"INTPNT_QO_TOL_DFEAS":1e-8,
"INTPNT_QO_TOL_INFEAS":1e-10,
"INTPNT_QO_TOL_MU_RED":1e-8,
"INTPNT_QO_TOL_NEAR_REL":1e+3,
"INTPNT_QO_TOL_PFEAS":1e-8,
"INTPNT_QO_TOL_REL_GAP":1e-8,
"INTPNT_TOL_DFEAS":1e-8,
"INTPNT_TOL_DSAFE":1e+0,
"INTPNT_TOL_INFEAS":1e-10,
"INTPNT_TOL_MU_RED":1e-16,
"INTPNT_TOL_PATH":1e-8,
"INTPNT_TOL_PFEAS":1e-8,
"INTPNT_TOL_PSAFE":1e+0,
"INTPNT_TOL_REL_GAP":1e-8,
"INTPNT_TOL_REL_STEP":9.999e-1,
"INTPNT_TOL_STEP_SIZE":1e-6,
"LOWER_OBJ_CUT":-1e+30,
"LOWER_OBJ_CUT_FINITE_TRH":-5e+29,
"MIO_DISABLE_TERM_TIME":-1e+0,
"MIO_MAX_TIME":-1e+0,
"MIO_MAX_TIME_APRX_OPT":6e+1,
"MIO_NEAR_TOL_ABS_GAP":0.0,
"MIO_NEAR_TOL_REL_GAP":1e-3,
"MIO_REL_GAP_CONST":1e-10,
"MIO_TOL_ABS_GAP":0.0,
"MIO_TOL_ABS_RELAX_INT":1e-5,
"MIO_TOL_FEAS":1e-6,
"MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT":0.0,
"MIO_TOL_REL_GAP":1e-4,
"MIO_TOL_X":1e-6,
"OPTIMIZER_MAX_TIME":-1e+0,
"PRESOLVE_TOL_ABS_LINDEP":1e-6,
"PRESOLVE_TOL_AIJ":1e-12,

```

```

        "PRESOLVE_TOL_REL_LINDEP":1e-10,
        "PRESOLVE_TOL_S":1e-8,
        "PRESOLVE_TOL_X":1e-8,
        "QCQO_REFORMULATE_REL_DROP_TOL":1e-15,
        "SEMIDEFINITE_TOL_APPROX":1e-10,
        "SIM_LU_TOL_REL_PIV":1e-2,
        "SIMPLEX_ABS_TOL_PIV":1e-7,
        "UPPER_OBJ_CUT":1e+30,
        "UPPER_OBJ_CUT_FINITE_TRH":5e+29
    },
    "sparam":{
        "BAS_SOL_FILE_NAME":"",
        "DATA_FILE_NAME":"examples/tools/data/lo1.mps",
        "DEBUG_FILE_NAME":"",
        "INT_SOL_FILE_NAME":"",
        "ITR_SOL_FILE_NAME":"",
        "MIO_DEBUG_STRING":"",
        "PARAM_COMMENT_SIGN":"%",
        "PARAM_READ_FILE_NAME":"",
        "PARAM_WRITE_FILE_NAME":"",
        "READ_MPS_BOU_NAME":"",
        "READ_MPS_OBJ_NAME":"",
        "READ_MPS_RAN_NAME":"",
        "READ_MPS_RHS_NAME":"",
        "SENSITIVITY_FILE_NAME":"",
        "SENSITIVITY_RES_FILE_NAME":"",
        "SOL_FILTER_XC_LOW":"",
        "SOL_FILTER_XC_UPR":"",
        "SOL_FILTER_XX_LOW":"",
        "SOL_FILTER_XX_UPR":"",
        "STAT_FILE_NAME":"",
        "STAT_KEY":"",
        "STAT_NAME":"",
        "WRITE_LP_GEN_VAR_NAME":"XMSKGEN"
    }
}
}
}

```

15.8 The Solution File Format

MOSEK provides several solution files depending on the problem type and the optimizer used:

- *basis solution file* (extension `.bas`) if the problem is optimized using the simplex optimizer or basis identification is performed,
- *interior solution file* (extension `.sol`) if a problem is optimized using the interior-point optimizer and no basis identification is required,
- *integer solution file* (extension `.int`) if the problem contains integer constrained variables.

All solution files have the format:

NAME	: <problem name>
PROBLEM STATUS	: <status of the problem>
SOLUTION STATUS	: <status of the solution>
OBJECTIVE NAME	: <name of the objective function>
PRIMAL OBJECTIVE	: <primal objective value corresponding to the solution>

(continues on next page)

(continued from previous page)

DUAL OBJECTIVE : <dual objective value corresponding to the solution>						
CONSTRAINTS						
INDEX	NAME	AT	ACTIVITY	LOWER LIMIT	UPPER LIMIT	DUAL LOWER
?	<name>	??	<a value>	<a value>	<a value>	<a value>
VARIABLES						
INDEX	NAME	AT	ACTIVITY	LOWER LIMIT	UPPER LIMIT	DUAL LOWER
?	<name>	??	<a value>	<a value>	<a value>	<a value>
↪ CONIC DUAL						
?	<name>	??	<a value>	<a value>	<a value>	<a value>
↪ <a value>						

In the example the fields ? and <> will be filled with problem and solution specific information. As can be observed a solution report consists of three sections, i.e.

- **HEADER** In this section, first the name of the problem is listed and afterwards the problem and solution status are shown. Next the primal and dual objective values are displayed.
- **CONSTRAINTS** For each constraint i of the form

$$l_i^c \leq \sum_{j=1}^n a_{ij}x_j \leq u_i^c, \quad (15.15)$$

the following information is listed:

- **INDEX**: A sequential index assigned to the constraint by **MOSEK**
- **NAME**: The name of the constraint assigned by the user.
- **AT**: The status of the constraint. In Table 15.4 the possible values of the status keys and their interpretation are shown.

Table 15.4: Status keys.

Status key	Interpretation
UN	Unknown status
BS	Is basic
SB	Is superbasic
LL	Is at the lower limit (bound)
UL	Is at the upper limit (bound)
EQ	Lower limit is identical to upper limit
**	Is infeasible i.e. the lower limit is greater than the upper limit.

- **ACTIVITY**: the quantity $\sum_{j=1}^n a_{ij}x_j^*$, where x^* is the value of the primal solution.
- **LOWER LIMIT**: the quantity l_i^c (see (15.15).)
- **UPPER LIMIT**: the quantity u_i^c (see (15.15).)
- **DUAL LOWER**: the dual multiplier corresponding to the lower limit on the constraint.
- **DUAL UPPER**: the dual multiplier corresponding to the upper limit on the constraint.
- **VARIABLES** The last section of the solution report lists information about the variables. This information has a similar interpretation as for the constraints. However, the column with the header **CONIC DUAL** is included for problems having one or more conic constraints. This column shows the dual variables corresponding to the conic constraints.

Example: lo1.sol

In Listing 15.7 we show the solution file for the lo1.opf problem.

Listing 15.7: An example of .sol file.

```

NAME          :
PROBLEM STATUS : PRIMAL_AND_DUAL_FEASIBLE
SOLUTION STATUS : OPTIMAL
OBJECTIVE NAME  : obj
PRIMAL OBJECTIVE : 8.33333333e+01
DUAL OBJECTIVE  : 8.33333332e+01

CONSTRAINTS
INDEX      NAME          AT ACTIVITY          LOWER LIMIT      UPPER LIMIT
↪      DUAL LOWER          DUAL UPPER
0          c1              EQ 3.00000000000000e+01      3.00000000e+01      3.
↪00000000e+01      -0.00000000000000e+00      -2.49999999741653e+00
1          c2              SB 5.33333333049187e+01      1.50000000e+01      NONE
↪          2.09159033069640e-10      -0.00000000000000e+00
2          c3              UL 2.49999999842049e+01      NONE              2.
↪50000000e+01      -0.00000000000000e+00      -3.33333332895108e-01

VARIABLES
INDEX      NAME          AT ACTIVITY          LOWER LIMIT      UPPER LIMIT
↪      DUAL LOWER          DUAL UPPER
0          x1              LL 1.67020427038537e-09      0.00000000e+00      NONE
↪          -4.49999999528054e+00      -0.00000000000000e+00
1          x2              LL 2.93510446211883e-09      0.00000000e+00      1.
↪00000000e+01      -2.16666666494915e+00      6.20868657679896e-10
2          x3              SB 1.49999999899424e+01      0.00000000e+00      NONE
↪          -8.79123177245553e-10      -0.00000000000000e+00
3          x4              SB 8.33333332273115e+00      0.00000000e+00      NONE
↪          -1.69795978848200e-09      -0.00000000000000e+00

```

Chapter 16

List of examples

List of examples shipped in the distribution of Fusion API for C++:

Table 16.1: List of distributed examples

File	Description
TrafficNetworkModel.cc	Demonstrates a traffic network problem as a conic quadratic problem (CQO)
alan.cc	A portfolio choice model <code>alan.gms</code> from the GAMS model library
baker.cc	A small bakery revenue maximization linear problem
breaksolver.cc	Shows how to break a long-running task
callback.cc	An example of data/progress callback
ceo1.cc	A simple conic exponential problem
cqo1.cc	A simple conic quadratic problem
diet.cc	Solving Stigler's Nutrition model <code>diet</code> from the GAMS model library
duality.cc	Shows how to access the dual solution
elastic.cc	Linear regression with elastic net. Demonstrates model parametrization.
facility_location.cc	Demonstrates a small one-facility location problem (CQO)
gp1.cc	A simple geometric program (GP) in conic form
lo1.cc	A simple linear problem
logistic.cc	Implements logistic regression and simple log-sum-exp (CEO)
lownerjohn_ellipsoids.cc	Computes the Lowner-John inner and outer ellipsoidal approximations of a polytope (SDO, Power Cone)
lpt.cc	Demonstrates how to solve the multi-processor scheduling problem and input an integer feasible point (MIP)
mico1.cc	A simple mixed-integer conic problem
milo1.cc	A simple mixed-integer linear problem
miointsol.cc	A simple mixed-integer linear problem with an initial guess
modelLib.cc	Library of implementations of basic functions
nearestcorr.cc	Solves the nearest correlation matrix problem (SDO, CQO)
opt_server_sync.cc	Uses MOSEK OptServer to solve an optimization problem synchronously
parameters.cc	Shows how to set optimizer parameters and read information items
portfolio_1_basic.cc	Portfolio optimization - basic Markowitz model
portfolio_2_frontier.cc	Portfolio optimization - efficient frontier
portfolio_3_impact.cc	Portfolio optimization - market impact costs
portfolio_4_transaction.cc	Portfolio optimization - transaction costs

continues on next page

Table 16.1 – continued from previous page

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

Chapter 17

Interface changes

The section shows interface-specific changes to the **MOSEK** Fusion API for C++ in version 9.3 compared to version 8. See the [release notes](#) for general changes and new features of the **MOSEK** Optimization Suite.

17.1 Backwards compatibility

- **Parametrized models (from version 9.2).** Introduced the possibility to parametrize *Fusion* models. See [Sec. 6.6](#) and [Sec. 7.9](#).
- **OptServer (from version 9.1).** It is possible to call the OptServer from *Fusion* with `M.solve(server, port)`.

Compared to version 8 there is a number of small changes in the *Fusion* API. Most of them should not affect standard applications of *Fusion* and very few should cause compilation errors.

- **Parameters.** Users who set parameters to tune the performance and numerical properties of the solver (termination criteria, tolerances, solving primal or dual, presolve etc.) are recommended to reevaluate such tuning. It may be that other, or default, parameter settings will be more beneficial in the current version. The hints in [Sec. 9](#) may be useful for some cases.
- Remove all Near problem and solution statuses i.e. `SolutionStatus.NearOptimal`, `SolutionStatus.NearCertificate`, `AccSolutionStatus.NearOptimal`, etc. See [Sec. 13.3.3](#).
- Some algebraic operators are more strict when it comes to exactly matching shapes, especially regarding shapes such as $1 \times n$, $n \times 1$ and n . The same applies to matching variable/expression and domain shapes. In some cases it may be necessary to explicitly reshape using the method `Variable.reshape` or `LinearDomain.withShape`.
- All shapes are specified with ordinary arrays instead of objects of class `Set`. The static methods of the class `Set` produce arrays as shape specifications.
- Replace `shape()` with `getShape()` and `size()` with `getSize()` in most places.
- Remove the option in `Expr.sum` to sum over a range of dimensions in a multidimensional expression.
- Remove the method `Constraint.add` and introduce `Constraint.update` (see [Sec. 7.10](#)).
- Rename `QConeDomain` to `ConeDomain`.
- `Variable` now extends `Expression`, so the number of method prototypes is reduced in some cases. A variable can be used everywhere an expression can.
- Expressions are evaluated lazily only when used in a constraint.
- **Semidefinite variables.** The preferred ways to declare semidefinite variables are:

```

M->variable(Domain::inPSDCone(3));           // A 3x3 PSD variable
M->variable(Domain::inPSDCone(3, 100));       // One hundred 3x3 PSD variables
                                              // arranged in shape [100, 3, 3]

```

17.2 Parameters

Added

- *intpntOrderGpNumSeeds*
- *logLocalInfo*
- *mioConicOuterApproximation*
- *mioFeaspumpLevel*
- *mioMaxNumRootCutRounds*
- *mioPropagateObjectiveConstraint*
- *mioSeed*
- *presolveMaxNumPass*
- *simSeed*

Removed

- *dataTolAij*
- *intpntNlMeritBal*
- *intpntNlTolDfeas*
- *intpntNlTolMuRed*
- *intpntNlTolNearRel*
- *intpntNlTolPfeas*
- *intpntNlTolRelGap*
- *intpntNlTolRelStep*
- *mioDisableTermTime*
- *mioNearTolAbsGap*
- *mioNearTolRelGap*
- *mioConstructSol*
- *mioMtUserCb*
- *opfMaxTermsPerLine*
- *readDataCompressed*
- *readDataFormat*
- *writeDataCompressed*
- *writeDataFormat*

17.3 Constants

Added

Removed

- xml
- mioHeuristicTime
- mioOptimizerTime
- mioConstructNumRoundings
- mioInitialSolution
- mioNearAbsgapSatisfied
- mioNearRelgapSatisfied
- mioSimMaxiterSetbacks
- hybrid
- worst
- geco
- nearDualFeas
- nearPrimAndDualFeas
- nearPrimFeas
- optimalPartition
- nearDualFeas
- nearDualInfeasCer
- nearIntegerOptimal
- nearOptimal
- nearPrimAndDualFeas
- nearPrimFeas
- nearPrimInfeasCer

Bibliography

- [AA95] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Math. Programming*, 71(2):221–245, 1995.
- [AGMeszarosX96] E. D. Andersen, J. Gondzio, Cs. Mészáros, and X. Xu. Implementation of interior point methods for large scale linear programming. In T. Terlaky, editor, *Interior-point methods of mathematical programming*, pages 189–252. Kluwer Academic Publishers, 1996.
- [ART03] E. D. Andersen, C. Roos, and T. Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Math. Programming*, February 2003.
- [AY96] E. D. Andersen and Y. Ye. Combining interior-point and pivoting algorithms. *Management Sci.*, 42(12):1719–1731, December 1996.
- [And09] Erling D. Andersen. The homogeneous and self-dual model and algorithm for linear optimization. Technical Report TR-1-2009, MOSEK ApS, 2009. URL: <http://docs.mosek.com/whitepapers/homolo.pdf>.
- [And13] Erling D. Andersen. On formulating quadratic functions in optimization models. Technical Report TR-1-2013, MOSEK ApS, 2013. Last revised 23-feb-2016. URL: <http://docs.mosek.com/whitepapers/qmodel.pdf>.
- [BKVH07] S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi. A Tutorial on Geometric Programming. *Optimization and Engineering*, 8(1):67–127, 2007. Available at http://www.stanford.edu/boyd/gp_tutorial.html.
- [Chvatal83] V. Chvátal. *Linear programming*. W.H. Freeman and Company, 1983.
- [GJ79] Michael R Gary and David S Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979.
- [GY05] Donald Goldfarb and Wotao Yin. Second-order cone programming methods for total variation-based image restoration. *SIAM Journal on Scientific Computing*, 27(2):622–645, 2005.
- [Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.
- [GK00] Richard C. Grinold and Ronald N. Kahn. *Active portfolio management*. McGraw-Hill, New York, 2 edition, 2000.
- [Naz87] J. L. Nazareth. *Computer Solution of Linear Programs*. Oxford University Press, New York, 1987.
- [PB15] Jaehyun Park and Stephen Boyd. A semidefinite programming method for integer convex quadratic minimization. *arXiv preprint arXiv:1504.07672*, 2015.
- [Pat03] Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM review*, 45(1):116–123, 2003.
- [Wol98] L. A. Wolsey. *Integer programming*. John Wiley and Sons, 1998.
- [BenTalN01] A. Ben-Tal and A. Nemirovski. *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. MPS/SIAM Series on Optimization. SIAM, 2001.

Response codes

`mioMode`, 276
`mioNodeOptimizer`, 276
`mioNodeSelection`, 276
`mioPerspectiveReformulate`, 276
`mioProbingLevel`, 276
`mioPropagateObjectiveConstraint`, 277
`mioRinsMaxNodes`, 277
`mioRootOptimizer`, 277
`mioRootRepeatPresolveLevel`, 277
`mioSeed`, 277
`mioVbDetectionLevel`, 278
`mtSpincount`, 278
`numThreads`, 278
`optimizer`, 278
`presolveEliminatorMaxFill`, 278
`presolveEliminatorMaxNumTries`, 279
`presolveLevel`, 279
`presolveLindepAbsWorkTrh`, 279
`presolveLindepRelWorkTrh`, 279
`presolveLindepUse`, 279
`presolveMaxNumPass`, 279
`presolveUse`, 280
`removeUnusedSolutions`, 280
`simBasisFactorUse`, 280
`simDegen`, 280
`simDualCrash`, 280
`simDualPhaseoneMethod`, 280
`simDualRestrictSelection`, 281
`simDualSelection`, 281
`simExploitDupvec`, 281
`simHotstart`, 281
`simHotstartLu`, 281
`simMaxIterations`, 281
`simMaxNumSetbacks`, 282
`simNonSingular`, 282
`simPrimalCrash`, 282
`simPrimalPhaseoneMethod`, 282
`simPrimalRestrictSelection`, 282
`simPrimalSelection`, 283
`simRefactorFreq`, 283
`simReformulation`, 283
`simSaveLu`, 283
`simScaling`, 283
`simScalingMethod`, 283
`simSeed`, 283
`simSolveForm`, 284
`simSwitchOptimizer`, 284
`writeLpFullObj`, 284
`writeLpLineWidth`, 284
`writeLpQuotedNames`, 284
`writeLpTermsPerLine`, 284
`String parameters`, 285
`basSolFileName`, 285
`dataFileName`, 285
`intSolFileName`, 285
`itrSolFileName`, 285
`remoteAccessToken`, 285
`writeLpGenVarName`, 285

