



MOSEK Optimization Toolbox for
MATLAB
Release 9.0.98

MOSEK ApS

15 July 2019

Contents

1	Introduction	1
1.1	Why the Optimization Toolbox for MATLAB?	2
2	Contact Information	3
3	License Agreement	4
4	Installation	6
4.1	Testing the installation	7
4.2	Troubleshooting	7
5	Design Overview	9
5.1	Modeling	9
5.2	“Hello World!” in MOSEK	9
6	Optimization Tutorials	11
6.1	Linear Optimization	11
6.2	Quadratic Optimization	14
6.3	Conic Quadratic Optimization	17
6.4	Power Cone Optimization	19
6.5	Conic Exponential Optimization	21
6.6	Semidefinite Optimization	22
6.7	Affine conic constraints (new)	24
6.8	Geometric Programming	27
6.9	Integer Optimization	29
6.10	Problem Modification and Reoptimization	31
7	Solver Interaction Tutorials	36
7.1	Accessing the solution	36
7.2	Errors and exceptions	39
7.3	Input/Output	41
7.4	Setting solver parameters	42
7.5	Retrieving information items	43
7.6	Progress and data callback	44
8	Debugging Tutorials	46
8.1	Understanding optimizer log	46
8.2	Addressing numerical issues	50
8.3	Debugging infeasibility	52
8.4	Python Console	56
9	Advanced Numerical Tutorials	59
9.1	Converting a quadratically constrained problem to conic form	59
9.2	Advanced hot-start	62
10	Technical guidelines	66
10.1	Integration with MATLAB	66
10.2	Names	67

10.3	Multithreading	67
10.4	The license system	67
11	Case Studies	69
11.1	Portfolio Optimization	69
11.2	Least Squares and Other Norm Minimization Problems	80
11.3	Robust linear Optimization	84
12	Problem Formulation and Solutions	97
12.1	Linear Optimization	97
12.2	Conic Optimization	100
12.3	Semidefinite Optimization	104
12.4	Quadratic and Quadratically Constrained Optimization	105
12.5	Affine Conic Constraints	106
13	Optimizers	108
13.1	Presolve	108
13.2	Linear Optimization	110
13.3	Conic Optimization - Interior-point optimizer	116
13.4	The Optimizer for Mixed-integer Problems	120
14	Additional features	125
14.1	Problem Analyzer	125
14.2	Automatic Repair of Infeasible Problems	126
14.3	Sensitivity Analysis	129
15	Toolbox API Reference	137
15.1	API conventions	137
15.2	Command Reference	137
15.3	Data Structures and Notation	145
15.4	Parameters grouped by topic	153
15.5	Parameters (alphabetical list sorted by type)	165
15.6	Response codes	204
15.7	Enumerations	222
15.8	Nonlinear interfaces (obsolete)	246
16	Supported File Formats	248
16.1	The LP File Format	249
16.2	The MPS File Format	254
16.3	The OPF Format	265
16.4	The CBF Format	274
16.5	The PTF Format	288
16.6	The Task Format	292
16.7	The JSON Format	293
16.8	The Solution File Format	300
17	List of examples	303
18	Interface changes	305
18.1	Backwards compatibility	305
18.2	New API	305
18.3	Parameters	305
18.4	Constants	306
18.5	Response Codes	308
	Bibliography	310
	Symbol Index	311
	Index	324

Chapter 1

Introduction

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

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

In order to obtain an overview of features in the **MOSEK** Optimization Suite consult the [product introduction](#) guide.

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

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

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

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

$$Ax - b \geq 0.$$

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

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

where \mathcal{K} is a *convex cone*. For example in 3 dimensions \mathcal{K} may correspond to an ice cream cone. The conic optimizer in **MOSEK** supports a number of different types of cones \mathcal{K} , which allows a surprisingly large number of nonlinear relations to be modeled, as described in the **MOSEK Modeling Cookbook**, while preserving the nice algorithmic and theoretical properties of linear optimization.

1.1 Why the Optimization Toolbox for MATLAB?

The Optimization Toolbox for MATLAB provides access to most of the functionality of **MOSEK** from a MATLAB environment. In addition the toolbox includes functions that replace functions from the MATLAB optimization toolbox available from MathWorks.

The Optimization Toolbox for MATLAB provides access to:

- Linear Optimization (LO)
- Conic Quadratic (Second-Order Cone) Optimization (CQO, SOCO)
- Power Cone Optimization
- Conic Exponential Optimization (CEO)
- Convex Quadratic and Quadratically Constrained Optimization (QO, QCQO)
- Semidefinite Optimization (SDO)
- Mixed-Integer Optimization (MIO)

as well as to additional functions for:

- problem analysis,
- sensitivity analysis,
- infeasibility diagnostics.

Chapter 2

Contact Information

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

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

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

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

Chapter 3

License Agreement

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

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

zlib

MOSEK includes the *zlib* library obtained from the [zlib website](#). The license agreement for *zlib* is shown in [Listing 3.1](#).

Listing 3.1: *zlib* license.

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

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

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

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

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

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

fplib

MOSEK includes the floating point formatting library developed by David M. Gay obtained from the [netlib website](#). The license agreement for *fplib* is shown in [Listing 3.2](#).

Listing 3.2: *fplib* license.

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

(continues on next page)

```
* 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](https://github.com/facebook/zstd). The license agreement for *Zstandard* is shown in [Listing 3.3](#).

Listing 3.3: *Zstandard* license.

```
BSD License

For Zstandard software

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

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

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

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

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

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

Chapter 4

Installation

In this section we discuss how to install and setup the **MOSEK** Optimization Toolbox for MATLAB.

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 Optimization Toolbox for MATLAB can be used with MATLAB version R2015a or newer on all 64-bit platforms.

Locating files in the MOSEK Optimization Suite

The relevant files of the Optimization Toolbox for MATLAB are organized as reported in [Table 4.1](#).

Table 4.1: Relevant files for the Optimization Toolbox for MATLAB.

Relative Path	Description	Label
<MSKHOME>/mosek/9.0/toolbox/r2015a	Toolbox	<TOOLBOXDIR>
<MSKHOME>/mosek/9.0/toolbox/r2015aom	Toolbox (without overloading)	<TOOLBOXOMDIR>
<MSKHOME>/mosek/9.0/toolbox/examples	Examples	<EXDIR>
<MSKHOME>/mosek/9.0/toolbox/data	Additional data	<MISCDIR>

where <MSKHOME> is the folder in which the **MOSEK** Optimization Suite has been installed.

Setting up the paths

To use Optimization Toolbox for MATLAB the path to the toolbox directory must be added via the `addpath` command in MATLAB. Use the command

```
addpath <MSKHOME>/mosek/9.0/toolbox/r2015a
```

or, if you do not want to overload functions such as *linprog* and *quadprog* from the MATLAB Optimization Toolbox with their **MOSEK** versions, then write

```
addpath <MSKHOME>/mosek/9.0/toolbox/r2015aom
```

On the Windows platform the relevant paths are

```
addpath <MSKHOME>\mosek\9.0\toolbox\r2015a
```

```
addpath <MSKHOME>\mosek\9.0\toolbox\r2015aom
```

Alternatively, the path to Optimization Toolbox for MATLAB may be set from the command line or it can be added to MATLAB permanently using the configuration file `startup.m` or from the FileSet Path menu item. We refer to MATLAB documentation for details.

4.1 Testing the installation

You can verify that Optimization Toolbox for MATLAB works by executing

```
mosekdiag
```

in MATLAB. This should produce a message similar to this:

```
mosekopt: /home/username/mosek/current/toolbox/r2015a/mosekopt.mexa64
Found MOSEK version : major(9), minor(0), revision(37)
mosekopt is working correctly.
```

If you only want to use Optimization Toolbox for MATLAB then warnings about non-availability of the command-line interface can be ignored.

More advanced debug information can be obtained with:

```
mosekopt('debug(10)')
```

4.2 Troubleshooting

Missing library files such as libmosek64.9.0.dylib or similar

If you are using Mac OS and get an error such as

```
Library not loaded: libmosek64.9.0.dylib
Referenced from:
/Users/.../mosek/9.0/toolbox/r2015a/mosekopt.mexmaci64
Reason: image not found.

Error in callmosek>doCall (line 224)
[res,sol] = mosekopt('minimize info',prob,param);
```

then most likely you did not run the **MOSEK** installation script `install.py` found in the `bin` directory. See also the [Installation guide](#) for details.

Windows, invalid MEX-file, cannot find shared libraries

If you are using Windows and get an error such as

```
Invalid MEX-file <MSKHOME>\Mosek\9.0\toolbox\r2015a\mosekopt.mexw64: The specified module
↳could not be found.
```

then MATLAB cannot load the **MOSEK** shared libraries, because the folder containing them is not in the system search path for DLLs. This can happen if **MOSEK** was installed manually and not using the MSI installer. The solution is to add the path `<MSKHOME>\mosek\9.0\tools\platform\<PLATFORM>\bin` to the system environment variable `PATH`. This can also be done per MATLAB session by using the `setenv` command in MATLAB before using **MOSEK**, for example:

```
setenv('PATH', [getenv('PATH') ';C:\Users\username\mosek\9.0\tools\platform\win64x86\bin']);
```

See also the [Installation guide](#) for details.

MATLAB String type is not supported

From R2017a MATLAB provides a new string type (with double quotes). It is not supported by the Optimization Toolbox for MATLAB and may cause confusing error messages. For example the following will give an error:

```
mosekopt("minimize", prob)

Return code - 1200 [MSK_RES_ERR_IN_ARGUMENT] [A function argument is incorrect.]
```

Always use old-fashioned character arrays (strings in single quotes).

MOSEK does not see new license file

If you updated your license file but **MOSEK** does not detect it then restart MATLAB. **MOSEK** is caching the license and it will not notice the change in the license file on disk.

Undefined Function or Variable *mosekopt*

If you get the MATLAB error message

```
Undefined function or variable 'mosekopt'
```

you have not added the path to the Optimization Toolbox for MATLAB correctly as described above.

Invalid MEX-file

For certain versions of Windows and MATLAB, the path to MEX files cannot contain spaces. Therefore, if you have installed **MOSEK** in C:\Program Files\Mosek and get a MATLAB error similar to:

```
Invalid MEX-file <MSKHOME>\Mosek\9.0\toolbox\r2015a\mosekopt.mexw64
```

try installing **MOSEK** in a different directory, for example C:\Users\<someuser>\.

Output Arguments not assigned

If you encounter an error like

```
Error in ==> mosekopt at 1
function [r,res] = mosekopt(cmd,prob,param,callback)

Output argument "r" (and maybe others) not assigned during call to
"C:\Users\username\mosek\9.0\toolbox\r2015a\mosekopt.m>mosekopt".
```

then a mismatch between 32 and 64 bit versions of **MOSEK** and MATLAB is likely. From MATLAB type

```
which mosekopt
```

which (for a successful installation) should point to a MEX file,

```
<MSKHOME>\mosek\9.0\toolbox\r2015a\mosekopt.mexw64
```

and not to a MATLAB .m file,

```
<MSKHOME>\mosek\9.0\toolbox\r2015a\mosekopt.m
```

Chapter 5

Design Overview

5.1 Modeling

Optimization Toolbox for MATLAB is an interface for specifying optimization problems directly in matrix form. It means that an optimization problem such as:

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

or

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

is specified by describing the matrices A , F , vectors b, c, g and a list of cones \mathcal{K} directly.

The main characteristics of this interface are:

- **Simplicity:** once the problem data is assembled in matrix form, it is straightforward to input it into the optimizer.
- **Exploiting sparsity:** data is entered in sparse format, enabling huge, sparse problems to be defined and solved efficiently.
- **Efficiency:** the API incurs almost no overhead between the user's representation of the problem and **MOSEK**'s internal one.

Optimization Toolbox for MATLAB does not aid with modeling. It is the user's responsibility to express the problem in **MOSEK**'s standard form, introducing, if necessary, auxiliary variables and constraints. See [Sec. 12](#) for the precise formulations of problems **MOSEK** solves.

5.2 “Hello World!” in MOSEK

Here we present the most basic workflow pattern when using Optimization Toolbox for MATLAB.

Create a prob structure

Optimization problems using Optimization Toolbox for MATLAB are specified using a *prob* structure that describes the numerical data of the problem. In most cases it consists of matrices of floating-point numbers.

Retrieving the solutions

When the problem is set up, the optimizer is invoked with the call to *mosekopt*. The call will return a response and a structure containing the solution to all variables. See further details in [Sec. 7](#).

We refer also to [Sec. 7](#) for information about more advanced mechanisms of interacting with the solver.

Source code example

Below is the most basic code sample that defines and solves a trivial optimization problem

$$\begin{array}{ll}\text{minimize} & x \\ \text{subject to} & 2.0 \leq x \leq 3.0.\end{array}$$

For simplicity the example does not contain any error or status checks.

Listing 5.1: “Hello World!” in MOSEK

```
prob.a = sparse(0,1) % 0 linear constraints, 1 variable
prob.c = [1.0]'      % Only objective coefficient
prob.blx= [2.0]'     % Lower bound(s) on variable(s)
prob.bux= [3.0]'     % Upper bound(s) on variable(s)

% Optimize
[r, res] = mosekopt('minimize', prob);

% Print answer
res.sol.itr.xx
```

Chapter 6

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.

6.1 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 .

6.1.1 Example LO1

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

$$\begin{aligned} \text{maximize} \quad & 3x_0 + 1x_1 + 5x_2 + 1x_3 \\ \text{subject to} \quad & 3x_0 + 1x_1 + 2x_2 = 30, \\ & 2x_0 + 1x_1 + 3x_2 + 1x_3 \geq 15, \\ & 2x_1 + 3x_3 \leq 25, \end{aligned} \tag{6.1}$$

under the bounds

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

Example: Linear optimization using `msklpopt`

A linear optimization problem such as (6.1) can be solved using the `msklpopt` function. The first step in solving the example is to setup the data for problem (6.1) i.e. the c , A , etc. Afterwards the problem is solved using an appropriate call to `msklpopt`.

Listing 6.1: Script implementing problem (6.1) using `msklpopt`.

```
function lol()

c    = [3 1 5 1]';
a    = [[3 1 2 0];[2 1 3 1];[0 2 0 3]];
blc  = [30 15 -inf]';
buc  = [30 inf 25 ]';
blx  = zeros(4,1);
bux  = [inf 10 inf inf]';

[res] = msklpopt(c,a,blc,buc,blx,bux,[],'maximize');
sol   = res.sol;

% Interior-point solution.

sol.itr.xx'    % x solution.
sol.itr.sux'   % Dual variables corresponding to buc.
sol.itr.slx'   % Dual variables corresponding to blx.

% Basic solution.

sol.bas.xx'    % x solution in basic solution.
```

Please note that:

- Infinite bounds are specified using `-inf` and `inf`. Moreover, using `[]` for `bux`, `buc`, `blx` or `blc` means there are no bounds of the corresponding type.
- Retrieving different solution types is discussed in Sec. 7.1.

Example: Linear optimization using `mosekopt`

The function `msklpopt` is just a wrapper around the `mosekopt`, which is the main interface to **MOSEK** and is the only choice for more complicated problems, for instance with conic constraints. We demonstrate how to solve (6.1) directly with `mosekopt`. The following MATLAB code demonstrate how to set up the `prob` structure for the example (6.1) and solve the problem using `mosekopt`.

Listing 6.2: Script implementing problem (6.1) using *mosekopt*.

```
function lo2()
clear prob;

% Specify the c vector.
prob.c = [3 1 5 1]';

% Specify a in sparse format.
subi = [1 1 1 2 2 2 2 3 3];
subj = [1 2 3 1 2 3 4 2 4];
valij = [3 1 2 2 1 3 1 2 3];

prob.a = sparse(subi,subj,valij);

% Specify lower bounds of the constraints.
prob.blc = [30 15 -inf]';

% Specify upper bounds of the constraints.
prob.buc = [30 inf 25 ]';

% Specify lower bounds of the variables.
prob.blx = zeros(4,1);

% Specify upper bounds of the variables.
prob.bux = [inf 10 inf inf]';

% Perform the optimization.
[r,res] = mosekopt('maximize',prob);

% Show the optimal x solution.
res.sol.bas.xx
```

Please note that

- A MATLAB structure named *prob* containing all the relevant problem data is defined.
- All fields of this structure are optional except *prob.a* which is required to be a **sparse** matrix. The dimension of this matrix determine the number of constraints and variables in the problem.
- Different parts of the solution can be accessed as described in Sec. 7.1.

Example: Linear optimization using *linprog*

MOSEK also provides a function *linprog* with a function of the same name from the MATLAB Optimization Toolbox. Consult Sec. 10.1 for details.

Listing 6.3: Script implementing problem (6.1) using *linprog*.

```
f = - [3 1 5 1]'; % minus because we maximize
A = [[-2 -1 -3 -1]; [0 2 0 3]];
b = [-15 25]';
Aeq = [3 1 2 0];
beq = 30;
l = zeros(4,1);
u = [inf 10 inf inf]';

% Example of setting options for linprog
% Get default options
opt = mskoptimset('');
% Turn on diagnostic output
opt = mskoptimset(opt,'Diagnostics','on');
```

(continues on next page)

```

% Set a MOSEK option, in this case turn basic identification off.
opt = mskoptimset(opt, 'MSK_IPAR_INTPNT_BASIS', 'MSK_OFF');
% Modify a MOSEK parameter with double value
opt = mskoptimset(opt, 'MSK_DPAR_INTPNT_TOL_INFEAS', 1e-12);

[x,fval,exitflag,output,lambda] = linprog(f,A,b,Aeq,beq,l,u,opt);

x
fval
exitflag
output
lambda

```

6.2 Quadratic Optimization

MOSEK can solve quadratic and quadratically constrained problems, as long as they are convex. This class of problems can be formulated as follows:

$$\begin{aligned}
 & \text{minimize} && \frac{1}{2}x^T Q^o x + c^T x + c^f \\
 & \text{subject to} && \begin{aligned} l_k^c &\leq \frac{1}{2}x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j &\leq u_k^c, & k = 0, \dots, m-1, \\ l_j^x &\leq x_j &\leq u_j^x, & j = 0, \dots, n-1. \end{aligned}
 \end{aligned} \tag{6.2}$$

Without loss of generality it is assumed that Q^o and Q^k are all symmetric because

$$x^T Q x = \frac{1}{2} x^T (Q + Q^T) x.$$

This implies that a non-symmetric Q can be replaced by the symmetric matrix $\frac{1}{2}(Q + Q^T)$.

The problem is required to be convex. More precisely, the matrix Q^o must be positive semi-definite and the k th constraint must be of the form

$$l_k^c \leq \frac{1}{2}x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j \tag{6.3}$$

with a negative semi-definite Q^k or of the form

$$\frac{1}{2}x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j \leq u_k^c.$$

with a positive semi-definite Q^k . This implies that quadratic equalities are *not* allowed. Specifying a non-convex problem will result in an error when the optimizer is called.

A matrix is positive semidefinite if all the eigenvalues of Q are nonnegative. An alternative statement of the positive semidefinite requirement is

$$x^T Q x \geq 0, \quad \forall x.$$

If the convexity (i.e. semidefiniteness) conditions are not met **MOSEK** will not produce reliable results or work at all.

6.2.1 Example: Quadratic Objective

We look at a small problem with linear constraints and quadratic objective:

$$\begin{aligned}
 & \text{minimize} && x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\
 & \text{subject to} && 1 \leq x_1 + x_2 + x_3 \\
 & && 0 \leq x.
 \end{aligned} \tag{6.4}$$

The matrix formulation of (6.4) has:

$$Q^o = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 0.2 & 0 \\ -1 & 0 & 2 \end{bmatrix}, c = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, A = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix},$$

with the bounds:

$$l^c = 1, u^c = \infty, l^x = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } u^x = \begin{bmatrix} \infty \\ \infty \\ \infty \end{bmatrix}$$

Please note the explicit $\frac{1}{2}$ in the objective function of (6.2) which implies that diagonal elements must be doubled in Q , i.e. $Q_{11} = 2$ even though 1 is the coefficient in front of x_1^2 in (6.4).

Using mosekopt

In Listing 6.4 we show how to use *mosekopt* to solve problem (6.4). This is the preferred way.

Listing 6.4: How to solve problem (6.4) using *mosekopt*.

```
function qo2()

clear prob;

% c vector.
prob.c = [0 -1 0]';

% Define the data.

% First the lower triangular part of q in the objective
% is specified in a sparse format. The format is:
%
%   Q(prob.qosubi(t),prob.qosubj(t)) = prob.qoval(t), t=1,...,4

prob.qosubi = [ 1  3  2  3]';
prob.qosubj = [ 1  1  2  3]';
prob.qoval  = [ 2 -1 0.2 2]';

% a, the constraint matrix
subi = ones(3,1);
subj = 1:3;
valij = ones(3,1);

prob.a = sparse(subi,subj,valij);

% Lower bounds of constraints.
prob.blc = [1.0]';

% Upper bounds of constraints.
prob.buc = [inf]';

% Lower bounds of variables.
prob.blx = sparse(3,1);

% Upper bounds of variables.
prob.bux = []; % There are no bounds.

[r,res] = mosekopt('minimize',prob);

% Display return code.
fprintf('Return code: %d\n',r);
```

(continues on next page)

```
% Display primal solution for the constraints.
res.sol.itr.xc'

% Display primal solution for the variables.
res.sol.itr.xx'
```

This sequence of commands looks much like the one that was used to solve the linear optimization example using *mosekopt* except that the definition of the Q matrix in `prob.mosekopt` requires that Q is specified in a sparse format. Indeed the vectors `qosubi`, `qosubj`, and `qoval` are used to specify the coefficients of Q in the objective using the principle

$$Q_{\text{qosubi}(t), \text{qosubj}(t)} = \text{qoval}(t), \text{ for } t = 1, \dots, \text{length}(\text{qosubi}).$$

An important observation is that due to Q being symmetric, only the lower triangular part of Q should be specified.

Using *mskqpopt*

In Listing 6.5 we show how to use *mskqpopt* to solve problem (6.4).

Listing 6.5: Function solving problem (6.4) using *mskqpopt*.

```
function qol()

% Set up Q.
q = [[2 0 -1]; [0 0.2 0]; [-1 0 2]];

% Set up the linear part of the problem.
c = [0 -1 0]';
a = ones(1,3);
blc = [1.0];
buc = [inf];
blx = sparse(3,1);
bux = [];

% Optimize the problem.
[res] = mskqpopt(q,c,a,blc,buc,blx,bux);

% Show the primal solution.
res.sol.itr.xx
```

It should be clear that the format for calling *mskqpopt* is very similar to calling *msklpopt* except that the Q matrix is included as the first argument of the call. Similarly, the solution can be inspected by viewing the `res.sol` field.

6.2.2 Example: Quadratic constraints

In this section we show how to solve a problem with quadratic constraints. Please note that quadratic constraints are subject to the convexity requirement (6.3).

Consider the problem:

$$\begin{aligned} & \text{minimize} && x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\ & \text{subject to} && 1 \leq x_1 + x_2 + x_3 - x_1^2 - x_2^2 - 0.1x_3^2 + 0.2x_1x_3, \\ & && x \geq 0. \end{aligned}$$

This is equivalent to

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Q^o x + c^T x \\ & \text{subject to} && \frac{1}{2}x^T Q^0 x + Ax \geq b, \\ & && x \geq 0, \end{aligned} \tag{6.5}$$

where

$$Q^o = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 0.2 & 0 \\ -1 & 0 & 2 \end{bmatrix}, c = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T, A = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, b = 1.$$

$$Q^0 = \begin{bmatrix} -2 & 0 & 0.2 \\ 0 & -2 & 0 \\ 0.2 & 0 & -0.2 \end{bmatrix}.$$

The linear parts and quadratic objective are set up the way described in the previous tutorial.

Setting up quadratic constraints

Listing 6.6: Script implementing problem (6.5).

```
function qcqo1()
clear prob;

% Specify the linear objective terms.
prob.c = [0, -1, 0];

% Specify the quadratic terms of the constraints.
prob.qcsubk = [1 1 1 1]';
prob.qcsubi = [1 2 3 3]';
prob.qcsubj = [1 2 3 1]';
prob.qcval = [-2.0 -2.0 -0.2 0.2]';

% Specify the quadratic terms of the objective.
prob.qosubi = [1 2 3 3]';
prob.qosubj = [1 2 3 1]';
prob.qoval = [2.0 0.2 2.0 -1.0]';

% Specify the linear constraint matrix
prob.a = [1 1 1];

% Specify the lower bounds
prob.blc = [1];
prob.blx = zeros(3,1);

[r,res] = mosekopt('minimize',prob);

% Display the solution.
fprintf('\nx:');
fprintf(' %-.4e',res.sol.itr.xx');
fprintf('\n||x||: %-.4e',norm(res.sol.itr.xx));
```

6.3 Conic Quadratic Optimization

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

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

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

MOSEK can solve conic quadratic optimization problems of the form

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

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

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

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

- Quadratic cone:

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

- Rotated quadratic cone:

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

For other types of cones supported by **MOSEK** see [Sec. 6.4](#), [Sec. 6.5](#), [Sec. 6.6](#). 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:

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

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

6.3.1 Example CQO1

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

$$\begin{aligned} & \text{minimize} && x_4 + x_5 + x_6 \\ & \text{subject to} && x_1 + x_2 + 2x_3 = 1, \\ & && x_1, x_2, x_3 \geq 0, \\ & && x_4 \geq \sqrt{x_1^2 + x_2^2}, \\ & && 2x_5x_6 \geq x_3^2 \end{aligned} \tag{6.6}$$

The linear constraints are specified as if the problem was a linear problem whereas the cones are specified using two index lists `cones.subptr` and `cones.sub` and list of cone-type identifiers `cones.type`. The elements of all the cones are listed in `cones.sub`, and `cones.subptr` specifies the index of the first element in `cones.sub` for each cone.

[Listing 6.7](#) demonstrates how to solve the example (6.6) using **MOSEK**.

Listing 6.7: Script implementing problem (6.6).

```
function cqo1()

clear prob;

[r, res] = mosekopt('symbcon');
% Specify the non-conic part of the problem.

prob.c = [0 0 0 1 1 1];
prob.a = sparse([1 1 2 0 0 0]);
prob.blc = 1;
```

(continues on next page)

(continued from previous page)

```
prob.buc = 1;
prob.blx = [0 0 0 -inf -inf -inf];
prob.bux = inf*ones(6,1);

% Specify the cones.

prob.cones.type = [res.symbcon.MSK_CT_QUAD, res.symbcon.MSK_CT_RQUAD];
prob.cones.sub = [4, 1, 2, 5, 6, 3];
prob.cones.subptr = [1, 4];
% The field 'type' specifies the cone types, i.e., quadratic cone
% or rotated quadratic cone. The keys for the two cone types are MSK_CT_QUAD
% and MSK_CT_RQUAD, respectively.
%
% The fields 'sub' and 'subptr' specify the members of the cones,
% i.e., the above definitions imply that
% x(4) >= sqrt(x(1)^2+x(2)^2) and 2 * x(5) * x(6) >= x(3)^2.

% Optimize the problem.

[r,res]=mosekopt('minimize',prob);

% Display the primal solution.

res.sol.itr.xx'
```

6.4 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. 6.3](#), [Sec. 6.5](#), [Sec. 6.6](#). Different cone types can appear together in one optimization problem.

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

6.4.1 Example POW1

Consider the following optimization problem which involves powers of variables:

$$\begin{aligned} & \text{maximize} && x^{0.2}y^{0.8} + z^{0.4} - x \\ & \text{subject to} && x + y + \frac{1}{2}z = 2, \\ & && x, y, z \geq 0. \end{aligned} \tag{6.7}$$

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:

$$\begin{aligned} & \text{maximize} && x_3 + x_4 - x_0 \\ & \text{subject to} && x_0 + x_1 + \frac{1}{2}x_2 = 2, \\ & && (x_0, x_1, x_3) \in \mathcal{P}_3^{0.2, 0.8}, \\ & && (x_2, x_5, x_4) \in \mathcal{P}_3^{0.4, 0.6}, \\ & && x_5 = 1. \end{aligned} \tag{6.8}$$

The linear constraints are specified as if the problem was a linear problem. The cone elements are specified using two index lists `cones.subptr` and `cones.sub`. Cone-type identifiers appear in `cones.type`, and the cone parameters α are in `cones.conepar`. The elements of all the cones are listed in `cones.sub`, and `cones.subptr` specifies the index of the first element in `cones.sub` for each cone.

[Listing 6.8](#) demonstrates how to solve the example (6.7) using **MOSEK**. The solution is

[0.06389298 0.78308564 2.30604283]

Listing 6.8: Script implementing problem (6.7).

```
function pow1()

clear prob;

[r, res] = mosekopt('symbcon');
% Specify the non-conic part of the problem.

prob.c = [-1 0 0 1 1 0];
prob.a = [1 1 0.5 0 0 0];
prob.blc = [2.0];
prob.buc = [2.0];
prob.blx = [-inf -inf -inf -inf -inf 1.0];
prob.bux = [ inf inf inf inf inf 1.0];

% Specify the cones.
prob.cones.type = [res.symbcon.MSK_CT_PPOW res.symbcon.MSK_CT_PPOW];
prob.cones.conepar = [0.2 0.4];
prob.cones.sub = [1 2 4 3 6 5];
prob.cones.subptr = [1 4];
% The field 'type' specifies the cone types, in this case power cones.
%
% The fields 'sub' and 'subptr' specify the members of the cones,
% i.e., the above definitions imply that
```

(continues on next page)

```

% (x(1), x(2), x(4)) and (x(3), x(6), x(5))
% are cones.
%
% The field 'conepar' specifies the alpha cone parameters (exponents)

% Optimize the problem.

[r,res]=mosekopt('maximize',prob);

% Display the primal solution.

res.sol.itr.xx'

```

6.5 Conic Exponential Optimization

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

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

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

MOSEK can solve conic optimization problems of the form

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

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

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

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

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

MOSEK also supports the dual exponential cone:

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

For other types of cones supported by **MOSEK** see [Sec. 6.3](#), [Sec. 6.4](#), [Sec. 6.6](#). 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:

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

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

6.5.1 Example CEO1

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

$$\begin{aligned}
& \text{minimize} && x_0 + x_1 \\
& \text{subject to} && x_0 + x_1 + x_2 = 1, \\
& && x_0 \geq x_1 \exp(x_2/x_1), \\
& && x_0, x_1 \geq 0.
\end{aligned} \tag{6.9}$$

The conic form of (6.9) is:

$$\begin{aligned} & \text{minimize} && x_0 + x_1 \\ & \text{subject to} && x_0 + x_1 + x_2 = 1, \\ & && (x_0, x_1, x_2) \in K_{\text{exp}}, \\ & && x \in \mathbb{R}^3. \end{aligned} \tag{6.10}$$

The linear constraints are specified as if the problem was a linear problem whereas the cones are specified using two index lists `cones.subptr` and `cones.sub` and list of cone-type identifiers `cones.type`. The elements of all the cones are listed in `cones.sub`, and `cones.subptr` specifies the index of the first element in `cones.sub` for each cone.

Listing 6.9 demonstrates how to solve the example (6.9) using **MOSEK**.

Listing 6.9: Script implementing problem (6.9).

```
function ceol()

clear prob;

[r, res] = mosekopt('symbcon');
% Specify the non-conic part of the problem.

prob.c = [1 1 0];
prob.a = sparse([1 1 1]);
prob.blc = 1;
prob.buc = 1;
prob.blx = [-inf -inf -inf];
prob.bux = [ inf inf inf];

% Specify the cones.

prob.cones.type = [res.symbcon.MSK_CT_PEXP];
prob.cones.sub = [1, 2, 3];
prob.cones.subptr = [1];
% The field 'type' specifies the cone types, in this case an exponential
% cone with key MSK_CT_PEXP.
%
% The fields 'sub' and 'subptr' specify the members of the cones,
% i.e., the above definitions imply that
% x(1) >= x(2)*exp(x(3)/x(2))

% Optimize the problem.

[r,res]=mosekopt('minimize',prob);

% Display the primal solution.

res.sol.itr.xx'
```

6.6 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

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

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 $\bar{A}_{i,j} \in \mathcal{S}^{r_j}$. We use standard notation for the matrix inner product, i.e., for $A, B \in \mathbb{R}^{m \times n}$ we have

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

6.6.1 Example SDO1

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

$$\begin{aligned} & \text{minimize} && \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \bar{X} \right\rangle + x_0 \\ & \text{subject to} && \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \bar{X} \right\rangle + x_0 &= 1, \\ & && \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \bar{X} \right\rangle + x_1 + x_2 &= 1/2, \\ & && x_0 \geq \sqrt{x_1^2 + x_2^2}, \quad \bar{X} \succeq 0, \end{aligned} \tag{6.11}$$

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

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

Listing 6.10 demonstrates how to solve this problem using **MOSEK**.

Listing 6.10: Code implementing problem (6.11).

```
function sdo1()
[r, res] = mosekopt('symbcon');

prob.c      = [1, 0, 0];

prob.bardim = [3];
prob.barc.subj = [1, 1, 1, 1, 1];
prob.barc.subk = [1, 2, 2, 3, 3];
prob.barc.subl = [1, 1, 2, 2, 3];
prob.barc.val  = [2.0, 1.0, 2.0, 1.0, 2.0];

prob.blc = [1, 0.5];
prob.buc = [1, 0.5];

% It is a good practice to provide the correct
% dimension of A as the last two arguments
% because it facilitates better error checking.
prob.a    = sparse([1, 2, 2], [1, 2, 3], [1, 1, 1], 2, 3);
```

(continues on next page)

```

prob.bara.subi = [1, 1, 1, 2, 2, 2, 2, 2, 2];
prob.bara.subj = [1, 1, 1, 1, 1, 1, 1, 1, 1];
prob.bara.subk = [1, 2, 3, 1, 2, 3, 2, 3, 3];
prob.bara.subl = [1, 2, 3, 1, 1, 1, 2, 2, 3];
prob.bara.val = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0];

prob.cones.type = [res.symbcon.MSK_CT_QUAD];
prob.cones.sub = [1, 2, 3];
prob.cones.subptr = [1];

[r,res] = mosekopt('minimize info',prob);

X = zeros(3);
X([1,2,3,5,6,9]) = res.sol.itr.barx;
X = X + tril(X,-1)';

x = res.sol.itr.xx;

```

The solution x is returned in `res.sol.itr.xx` and the numerical values of \bar{X}_j are returned in `res.sol.barx`; the lower triangular part of each \bar{X}_j is stacked column-by-column into an array, and each array is then concatenated forming a single array `res.sol.itr.barx` representing $\bar{X}_1, \dots, \bar{X}_p$. Similarly, the dual semidefinite variables \bar{S}_j are recovered through `res.sol.itr.bars`.

6.7 Affine conic constraints (new)

Optimization Toolbox for MATLAB can solve conic optimization problems in another format:

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

where $F \in \mathbb{R}^{k \times n}$ and $g \in \mathbb{R}^k$ specify an *affine conic constraint* of length (dimension) k . Usually \mathcal{K} will be a product of basic cones corresponding to individual constraints.

In this tutorial we demonstrate how to use the affine conic format. It supports all types of basic cones available in **MOSEK** and can be combined with semidefinite variables as in [Sec. 6.6](#).

6.7.1 Example AFFCO1

Consider the following simple optimization problem:

$$\begin{aligned}
& \text{maximize} && x_1^{1/3} + (x_1 + x_2 + 0.1)^{1/4} \\
& \text{subject to} && (x_1 - 0.5)^2 + (x_2 - 0.6)^2 \leq 1, \\
& && x_1 - x_2 \leq 1.
\end{aligned} \tag{6.12}$$

Adding auxiliary variables we convert this problem into an equivalent conic form:

$$\begin{aligned}
& \text{maximize} && t_1 + t_2 \\
& \text{subject to} && (1, x_1 - 0.5, x_2 - 0.6) \in \mathcal{Q}^3, \\
& && (x_1, 1, t_1) \in \mathcal{P}_3^{1/3, 2/3}, \\
& && (x_1 + x_2 + 0.1, 1, t_2) \in \mathcal{P}_3^{1/4, 3/4}, \\
& && x_1 - x_2 \leq 1.
\end{aligned} \tag{6.13}$$

Note that each of the vectors constrained to a cone is in a natural way an affine combination of the problem variables.

We first set up the linear part of the problem, including the number of variables, objective and all bounds precisely as in [Sec. 6.1](#). Cones will be defined using the `cones` structure. We construct the matrices F, g for each of the three cones. For example, the constraint $(1, x_1 - 0.5, x_2 - 0.6) \in \mathcal{Q}^3$ is

written in matrix form as

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -0.5 \\ -0.6 \end{bmatrix} \in \mathcal{Q}^3.$$

Below we set up the matrices and define the cone type as a quadratic cone of length 3:

```
% The quadratic cone
FQ = sparse([zeros(1,4); speye(2) zeros(2,2)]);
gQ = [1 -0.5 -0.6]';
cQ = [res.symbcon.MSK_CT_QUAD 3];
```

Next we demonstrate how to do the same for the second of the power cone constraints. Its affine representation is:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 1 \\ 0 \end{bmatrix} \in \mathcal{P}_3^{1/4, 3/4}.$$

The power cone is defined by its type, length, number of additional parameters (always equal to 2) and the exponents $\alpha, 1 - \alpha$ appearing in the cone definition. In fact any pair of positive real numbers *proportional* to $(\alpha, 1 - \alpha)$ may be used. They will be normalized to add up to 1:

```
% The power cone for (x_1+x_2+0.1, 1, t_2) \in POW3^(1/4, 3/4)
FP2 = sparse([1 1 zeros(1,2); zeros(1,4); zeros(1,2) 0 1]);
gP2 = [0.1 1 0]';
cP2 = [res.symbcon.MSK_CT_PPOW 3 2 1.0 3.0];
```

Once affine conic descriptions of all cones are ready it remains to stack them vertically into the matrix F and vector g and concatenate the cone descriptions in one list. Below is the full code for problem (6.13).

Listing 6.11: Script implementing conic version of problem (6.12).

```
function affco1()

[rcode, res] = mosekopt('symbcon echo(0)');
prob = [];

% Variables [x1; x2; t1; t2]
prob.c = [0, 0, 1, 1];

% Linear inequality x_1 - x_2 <= 1
prob.a = sparse([1, -1, 0, 0]);
prob.buc = 1;
prob.blc = [];

% The quadratic cone
FQ = sparse([zeros(1,4); speye(2) zeros(2,2)]);
gQ = [1 -0.5 -0.6]';
cQ = [res.symbcon.MSK_CT_QUAD 3];

% The power cone for (x_1, 1, t_1) \in POW3^(1/3, 2/3)
FP1 = sparse([1 0 zeros(1,2); zeros(1,4); zeros(1,2) 1 0]);
gP1 = [0 1 0]';
cP1 = [res.symbcon.MSK_CT_PPOW 3 2 1/3 2/3];

% The power cone for (x_1+x_2+0.1, 1, t_2) \in POW3^(1/4, 3/4)
FP2 = sparse([1 1 zeros(1,2); zeros(1,4); zeros(1,2) 0 1]);
gP2 = [0.1 1 0]';
```

(continues on next page)

(continued from previous page)

```

cP2 = [res.symbcon.MSK_CT_PPOW 3 2 1.0 3.0];

% All cones
prob.f = [FQ; FP1; FP2];
prob.g = [gQ; gP1; gP2];
prob.cones = [cQ cP1 cP2];

[r, res] = mosekopt('maximize', prob);

res.sol.itr.pobjval
res.sol.itr.xx(1:2)

```

6.7.2 Example AFFCO2

Consider the following simple linear dynamical system. A point in \mathbb{R}^n moves along a trajectory given by $z(t) = z(0) \exp(At)$, where $z(0)$ is the starting position and $A = \mathbf{Diag}(a_1, \dots, a_n)$ is a diagonal matrix with $a_i < 0$. Find the time after which $z(t)$ is within euclidean distance d from the origin. Denoting the coordinates of the starting point by $z(0) = (z_1, \dots, z_n)$ we can write this as an optimization problem in one variable t :

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && \sqrt{\sum_i (z_i \exp(a_i t))^2} \leq d, \end{aligned}$$

which can be cast into conic form as:

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && (d, z_1 y_1, \dots, z_n y_n) \in \mathcal{Q}^{n+1}, \\ & && (y_i, 1, a_i t) \in K_{\text{exp}}, \quad i = 1, \dots, n, \end{aligned} \tag{6.14}$$

with variable vector $x = [t, y_1, \dots, y_n]^T$.

We assemble all conic constraints in the form

$$Fx + g \in \mathcal{Q}^{n+1} \times (K_{\text{exp}})^n.$$

For the conic quadratic constraint this representation is

$$\begin{bmatrix} 0 & 0_n^T \\ 0_n & \mathbf{Diag}(z_1, \dots, z_n) \end{bmatrix} \begin{bmatrix} t \\ y \end{bmatrix} + \begin{bmatrix} d \\ 0_n \end{bmatrix} \in \mathcal{Q}^{n+1}.$$

For the i -th exponential cone we have

$$\begin{bmatrix} 0 & e_i^T \\ 0 & 0_n \\ a_i & 0_n \end{bmatrix} \begin{bmatrix} t \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \in K_{\text{exp}},$$

where e_i denotes a vector of length n with a single 1 in position i .

Listing 6.12: Script implementing problem (6.14).

```

function t = firstHittingTime(n, z, a, d)

[r, res] = mosekopt('symbcon echo(0)');
prob = [];

% Variables [t, y1, ..., yn]
prob.a = sparse(0, n+1);
prob.c = [1 zeros(1,n)];

% Quadratic cone
FQ = diag([0; z]);

```

(continues on next page)

```

gQ = [d; zeros(n,1)];

% All exponential cones
FE = sparse([1:3:3*n    3:3:3*n], ...
            [2:n+1     ones(1,n)], ...
            [ones(1,n)  a']);
gE = repmat([0; 1; 0], n, 1);

% Assemble input data
prob.f = [FQ; FE];
prob.g = [gQ; gE];
prob.cones = [res.symbcon.MSK_CT_QUAD n+1 repmat([res.symbcon.MSK_CT_PEXP 3], 1, n)];

% Solve
[r, res] = mosekopt('minimize', prob);
t = res.sol.itr.xx(1)

```

6.8 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

$$\begin{aligned}
 & \text{minimize} && f_0(x) \\
 & \text{subject to} && f_i(x) \leq 1, \quad i = 1, \dots, m, \\
 & && x_j > 0, \quad j = 1, \dots, n,
 \end{aligned} \tag{6.15}$$

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

$$\log\left(\sum_k \exp(a_k^T y + b_k)\right) \leq 0 \tag{6.16}$$

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.

6.8.1 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 :

$$\begin{aligned}
 & \text{maximize} && hwd \\
 & \text{subject to} && 2(hw + hd) \leq A_{\text{wall}}, \\
 & && wd \leq A_{\text{floor}}, \\
 & && \alpha \leq h/w \leq \beta, \\
 & && \gamma \leq d/w \leq \delta.
 \end{aligned} \tag{6.17}$$

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 (6.17) becomes

$$\begin{aligned} & \text{maximize} && x + y + z \\ & \text{subject to} && \log(\exp(x + y + \log(2/A_{\text{wall}})) + \exp(x + z + \log(2/A_{\text{wall}}))) \leq 0, \\ & && y + z \leq \log(A_{\text{floor}}), \\ & && \log(\alpha) \leq x - y \leq \log(\beta), \\ & && \log(\gamma) \leq z - y \leq \log(\delta). \end{aligned} \tag{6.18}$$

Next, we demonstrate how to implement a log-sum-exp constraint (6.16). It can be written as:

$$\begin{aligned} u_k &\geq \exp(a_k^T y + b_k), \quad (\text{equiv. } (u_k, 1, a_k^T y + b_k) \in K_{\text{exp}}), \\ \sum_k u_k &= 1. \end{aligned} \tag{6.19}$$

This presentation requires one extra variable u_k for each monomial appearing in the original posynomial constraint. It is natural to express the cone membership using an affine conic constraint (see Sec. 6.7). In this case:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ \log(2/A_{\text{wall}}) \\ 0 \\ 1 \\ \log(2/A_{\text{wall}}) \end{bmatrix} \in K_{\text{exp}} \times K_{\text{exp}}.$$

We can now use this function to assemble all constraints in the model. The linear part of the problem is entered as in Sec. 6.1.

Listing 6.13: Source code solving problem (6.18).

```
[r,res] = mosekopt('symcon');

% Input data
Awall = 200;
Afloor = 50;
alpha = 2;
beta = 10;
gamma = 2;
delta = 10;

% Objective
prob = [];
prob.c = [1, 1, 1, 0, 0]';

% Linear constraints:
% [ 0  0  0  1  1 ]      == 1
% [ 0  1  1  0  0 ]      <= log(Afloor)
% [ 1 -1  0  0  0 ]      in [log(alpha), log(beta)]
% [ 0 -1  1  0  0 ]      in [log(gamma), log(delta)]
%
prob.a = [ 0  0  0  1  1;
          0  1  1  0  0;
          1 -1  0  0  0;
          0 -1  1  0  0];

prob.blc = [ 1; -inf;          log(alpha); log(gamma) ];
prob.buc = [ 1; log(Afloor); log(beta); log(delta) ];

prob.blx = [ -inf; -inf; -inf; -inf; -inf];
```

(continues on next page)

```

prob.bux = [ inf; inf; inf; inf; inf];

% The conic part FX+g \in Kexp x Kexp
%   x  y  z  u  v
% [ 0  0  0  1  0 ]    0
% [ 0  0  0  0  0 ]    1           in Kexp
% [ 1  1  0  0  0 ]    log(2/Awall)
%
% [ 0  0  0  0  1 ]    0
% [ 0  0  0  0  0 ]    1           in Kexp
% [ 1  0  1  0  0 ] + log(2/Awall)
%
%
prob.f = sparse([0 0 0 1 0;
                 0 0 0 0 0;
                 1 1 0 0 0;
                 0 0 0 0 1;
                 0 0 0 0 0;
                 1 0 1 0 0]);

prob.g = [ 0; 1; log(2/Awall); 0; 1; log(2/Awall)];

prob.cones = [ res.symbcon.MSK_CT_PEXP, 3, res.symbcon.MSK_CT_PEXP, 3 ];

% Optimize and print results
[r,res]=mosekopt('maximize',prob);
exp(res.sol.itr.xx(1:3))

```

6.9 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.

6.9.1 Example MILO1

We use the example

$$\begin{aligned}
 & \text{maximize} && x_0 + 0.64x_1 \\
 & \text{subject to} && 50x_0 + 31x_1 \leq 250, \\
 & && 3x_0 - 2x_1 \geq -4, \\
 & && x_0, x_1 \geq 0 \quad \text{and integer}
 \end{aligned} \tag{6.20}$$

to demonstrate how to set up and solve a problem with integer variables. It has the structure of a linear optimization problem (see [Sec. 6.1](#)) 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.

The complete source for the example is listed in [Listing 6.14](#).

Listing 6.14: How to solve problem (6.20).

```

function milo1()
clear prob
prob.c = [1 0.64];
prob.a = [[50 31];[3 -2]];
prob.blc = [-inf -4];
prob.buc = [250 inf];
prob.blx = [0 0];

```

(continues on next page)

```

prob.bux      = [inf inf];

% Specify indexes of variables that are integer
% constrained.

prob.ints.sub = [1 2];

% Optimize the problem.
[r,res] = mosekopt('maximize',prob);

try
    % Display the optimal solution.
    res.sol.int
    res.sol.int.xx'
catch
    fprintf('MSKERROR: Could not get solution')
end

```

Please note that compared to a linear optimization problem with no integer-constrained variables:

- The `prob.ints.sub` field is used to specify the indexes of the variables that are integer-constrained.
- The optimal integer solution is returned in the `res.sol.int` MATLAB structure.

MOSEK also provides a wrapper for the `intlinprog` function found in the MATLAB optimization toolbox. This function solves linear problems with integer variables; see the reference section for details.

6.9.2 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 `"MSK_IINF_MIO_CONSTRUCT_SOLUTION"` and `"MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ"`, and via the `Construct solution objective` entry in the log. We concentrate on a simple example below.

$$\begin{aligned}
 &\text{maximize} && 7x_0 + 10x_1 + x_2 + 5x_3 \\
 &\text{subject to} && x_0 + x_1 + x_2 + x_3 \leq 2.5 \\
 & && x_0, x_1, x_2 \in \mathbb{Z} \\
 & && x_0, x_1, x_2, x_3 \geq 0
 \end{aligned} \tag{6.21}$$

Solution values can be set using the appropriate fields in the problem structure.

Listing 6.15: Implementation of problem (6.21) specifying an initial solution.

```

% Specify start guess for the integer variables.
prob.sol.int.xx = [1 1 0 nan]';

```

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      : 1.9500000000000e+01
```

The same information can be obtained from the API:

Listing 6.16: Retrieving information about usage of initial solution

```

res.info.MSK_IINF_MIO_CONSTRUCT_SOLUTION
res.info.MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ

```

6.9.3 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:

$$\begin{aligned} & \text{minimize} && x^2 + y^2 \\ & \text{subject to} && x \geq e^y + 3.8, \\ & && x, y \text{ integer.} \end{aligned} \tag{6.22}$$

The canonical conic formulation of (6.22) suitable for Optimization Toolbox for MATLAB is

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && (t, x, y) \in \mathcal{Q}^3 && (t \geq \sqrt{x^2 + y^2}) \\ & && (x - 3.8, 1, y) \in K_{\text{exp}} && (x - 3.8 \geq e^y) \\ & && x, y \text{ integer,} \\ & && t \in \mathbb{R}. \end{aligned} \tag{6.23}$$

Listing 6.17: Implementation of problem (6.23).

```
[rcode, res] = mosekopt('symbcon echo(0)');
symbcon = res.symbcon;
clear prob

% The full variable is [t; x; y]
prob.c = [1 0 0];
prob.a = sparse(0,3); % No constraints

% Conic part of the problem
prob.f = sparse([ eye(3);
                 0 1 0;
                 0 0 0;
                 0 0 1 ]);
prob.g = [0 0 0 -3.8 1 0]';
prob.cones = [symbcon.MSK_CT_QUAD 3 symbcon.MSK_CT_PEXP 3];

% Specify indexes of variables that are integers
prob.ints.sub = [2 3];

% Optimize the problem.
[r,res] = mosekopt('minimize',prob);

try
    res.sol.int.xx(2:3)
catch
    fprintf('MSKERROR: Could not get solution')
end
```

Note that the conic constraints are described using the format $Fx + g \in \mathcal{K}$, that is as *affine conic constraints*. See Sec. 6.7 for details.

Error and solution status handling were omitted for readability.

6.10 Problem Modification and Reoptimization

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

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

- add/remove,

- coefficient modifications,
- bounds modifications.

Especially removing variables and constraints can be costly. Special care must be taken with respect to constraints and variable indexes that may be invalidated.

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. This special case is discussed in [Sec. 14.3](#).

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

6.10.1 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:

$$\begin{aligned}
 & \text{maximize} && 1.5x_0 &+& 2.5x_1 &+& 3.0x_2 \\
 & \text{subject to} && 2x_0 &+& 4x_1 &+& 3x_2 &\leq 100000, \\
 & && 3x_0 &+& 2x_1 &+& 3x_2 &\leq 50000, \\
 & && 2x_0 &+& 3x_1 &+& 2x_2 &\leq 60000,
 \end{aligned} \tag{6.24}$$

and

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

Code in [Listing 6.18](#) loads and solves this problem.

Listing 6.18: Setting up and solving problem (6.24)

```

% Specify the c vector.
prob.c = [1.5 2.5 3.0]';

% Specify a in sparse format.
subi = [1 1 1 2 2 2 3 3 3];
subj = [1 2 3 1 2 3 1 2 3];
valij = [2 4 3 3 2 3 2 3 2];

prob.a = sparse(subi,subj,valij);

% Specify lower bounds of the constraints.
prob.blc = [-inf -inf -inf]';

% Specify upper bounds of the constraints.
prob.buc = [100000 50000 60000]';

```

(continues on next page)

(continued from previous page)

```
% Specify lower bounds of the variables.
prob.blx = zeros(3,1);

% Specify upper bounds of the variables.
prob.bux = [inf inf inf]';

% Perform the optimization.
param.MSK_IPAR_OPTIMIZER = 'MSK_OPTIMIZER_FREE_SIMPLEX';
[r,res] = mosekopt('maximize',prob,param);

% Show the optimal x solution.
res.sol.bas.xx
```

6.10.2 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 $a_{0,0} = 3$, which is done by directly modifying the A matrix of the problem, as shown below.

```
prob.a(1,1) = 3.0;
```

The problem now has the form:

$$\begin{array}{llllll} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \quad (6.25)$$

and

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

After this operation we can reoptimize the problem.

6.10.3 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 6.19](#)

Listing 6.19: How to add a new variable (column)

```
prob.c = [prob.c; 1.0];
prob.a = [prob.a, sparse([4.0 0.0 1.0]')];
prob.blx = [prob.blx; 0.0];
prob.bux = [prob.bux; inf];
```

After this operation the new problem is:

$$\begin{array}{llllllll} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 60000, \end{array} \quad (6.26)$$

and

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

6.10.4 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 6.20: Adding a new constraint.

```
prob.a      = [prob.a; sparse([1.0 2.0 1.0 1.0])];
prob.blc    = [prob.blc; -inf];
prob.buc    = [prob.buc; 30000.0];
```

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

6.10.5 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:

$$\begin{aligned}
 &\text{maximize} && 1.5x_0 &+& 2.5x_1 &+& 3.0x_2 &+& 1.0x_3 \\
 &\text{subject to} && 3x_0 &+& 4x_1 &+& 3x_2 &+& 4x_3 &\leq 80000, \\
 & && 3x_0 &+& 2x_1 &+& 3x_2 && &\leq 40000, \\
 & && 2x_0 &+& 3x_1 &+& 2x_2 &+& 1x_3 &\leq 50000, \\
 & && x_0 &+& 2x_1 &+& x_2 &+& x_3 &\leq 22000.
 \end{aligned} \tag{6.27}$$

In this case all we need to do is redefine the upper bound vector for the constraints, as shown in the next listing.

Listing 6.21: Change constraint bounds.

```
prob.buc    = [80000 40000 50000 22000]';
prob.sol    = res.sol;
[r,res] = mosekopt('maximize',prob,param);
res.sol.bas.xx
```

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

6.10.6 Advanced hot-start

In order to exploit the possibility of hot-starting the simplex algorithms it is necessary to pass the old basic solution when the modified problem is re-optimized. Without this operation the optimizer will simply start from scratch. Any subset of the basic solution may be provided, but to achieve the best results all fields of `res.sol.bas` should be present, that is `xx,xc,y,slx,sux,slc,suc,skx,skc`.

Listing 6.22: Passing the full basic solution.

```
% Reoptimize with changed coefficient
% Use previous solution to perform very simple hot-start.
% This part can be skipped, but then the optimizer will start
% from scratch on the new problem, i.e. without any hot-start.
prob.sol = [];
prob.sol.bas = res.sol.bas;
[r,res] = mosekopt('maximize',prob,param);
res.sol.bas.xx
```

If the dimensions of the problem (number of variables, constraints) have changed, the lengths of all fields have to be adjusted to be compatible with the reformulated problem. For example, here is an adjustment when adding a new variable:

Listing 6.23: Adjusting lengths in the solution fields related to variables.

```
% Reoptimize with a new variable and hot-start
% All parts of the solution must be extended to the new dimensions.
prob.sol = [];
prob.sol.bas = res.sol.bas;
prob.sol.bas.xx = [prob.sol.bas.xx; 0.0];
prob.sol.bas.slx = [prob.sol.bas.slx; 0.0];
prob.sol.bas.sux = [prob.sol.bas.sux; 0.0];
prob.sol.bas.skx = [prob.sol.bas.skx; 'UN'];
[r,res] = mosekopt('maximize',prob,param);
res.sol.bas.xx
```

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.

A more advanced discussion of hot-start is presented in [Sec. 9.2](#).

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 7

Solver Interaction Tutorials

In this section we cover the interaction with the solver.

7.1 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.

7.1.1 Solver termination

The optimizer provides a **response code** of type *rescode*, relevant for error handling. It indicates if any errors occurred in any phase of optimization (including processing input data). It will also indicate system-related errors (such as an out of memory error, licensing error etc.). Finally, it will also indicate if the optimizer terminated correctly, but for a non-standard reason, for example because it reached a time limit or met another criterion set by the user. Such termination codes are not errors. The expected value for a typical successful optimization without any special settings is *"MSK_RES_OK"*.

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. 7.3](#).

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

7.1.2 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 7.1: Types of solutions available from **MOSEK**

	Simplex optimizer	Interior-point optimizer	Mixed-integer optimizer
Linear problem	<code>res.sol.bas</code>	<code>res.sol.itr</code>	
Nonlinear continuous problem		<code>res.sol.itr</code>	
Problem with integer variables			<code>res.sol.int</code>

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

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

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

7.1.3 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 (*prosta*) 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 *"MSK_PRO_STA_PRIM_AND_DUAL_FEAS"*.
- **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 (*solsta*) provides the information about what the solution values actually contain. The most important broad categories of values are:

- **optimal** (*"MSK_SOL_STA_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.

Problem and solution status can be found in the fields *prosta* and *solsta* of a solution structure *solution*, for instance *res.sol.itr.prosta*, *res.sol.itr.solsta* for the interior-point solution.

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 7.2: Continuous problems (solution status for interior-point and basic solution)

Outcome	Problem status	Solution status
Optimal	<i>"MSK_PRO_STA_PRIM_AND_DUAL_FEAS"</i>	<i>"MSK_SOL_STA_OPTIMAL"</i>
Primal infeasible	<i>"MSK_PRO_STA_PRIM_INFEAS"</i>	<i>"MSK_SOL_STA_PRIM_INFEAS_CER"</i>
Dual infeasible (unbounded)	<i>"MSK_PRO_STA_DUAL_INFEAS"</i>	<i>"MSK_SOL_STA_DUAL_INFEAS_CER"</i>
Uncertain (stall, numerical issues, etc.)	<i>"MSK_PRO_STA_UNKNOWN"</i>	<i>"MSK_SOL_STA_UNKNOWN"</i>

Table 7.3: Integer problems (solution status for integer solution, others undefined)

Outcome	Problem status	Solution status
Integer optimal	<i>"MSK_PRO_STA_PRIM_FEAS"</i>	<i>"MSK_SOL_STA_INTEGER_OPTIMAL"</i>
Infeasible	<i>"MSK_PRO_STA_PRIM_INFEAS"</i>	<i>"MSK_SOL_STA_UNKNOWN"</i>
Integer feasible point	<i>"MSK_PRO_STA_PRIM_FEAS"</i>	<i>"MSK_SOL_STA_PRIM_FEAS"</i>
No conclusion	<i>"MSK_PRO_STA_UNKNOWN"</i>	<i>"MSK_SOL_STA_UNKNOWN"</i>

7.1.4 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:

- `res.sol.itr.pobjval`, `res.sol.itr.dobjval` — the primal and dual objective value.
- `res.sol.itr.xx` — solution values for the variables.
- `res.sol.itr.y`, `res.sol.itr.slx` and so on — dual values for the linear constraints

and many other fields of the `solution` structure (replace `itr` with `bas` or `int` for other solution types). Note that if the optimization failed then the `res.sol` field may not exist and attempting to access it will cause an error.

7.1.5 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 7.1: Sample framework for checking optimization result.

```
function response(inputfile)

cmd = sprintf('read(%s)', inputfile)
% In this example we read the problem from file
[r, res] = mosekopt(cmd)

% Read was successful
if strcmp(res.rcodestr, 'MSK_RES_OK')

    prob = res.prob;
    param = []

    % (Optionally) Uncomment the next line to get solution status Unknown
    % param.MSK_IPAR_INTPNT_MAX_ITERATIONS = 1

    % Perform the optimization.
    [r, res] = mosekopt('minimize', prob, param);

    % Expected result: The solution status of the interior-point solution is optimal.

    % Check if we have non-error termination code or OK
    if isempty(strfind(res.rcodestr, 'MSK_RES_ERR'))

        solsta = strcat('MSK_SOL_STA_', res.sol.itr.solsta)

        if strcmp(solsta, 'MSK_SOL_STA_OPTIMAL')
            fprintf('An optimal interior-point solution is located:\n');
            res.sol.itr.xx

        elseif strcmp(solsta, 'MSK_SOL_STA_DUAL_INFEAS_CER')
            fprintf('Dual infeasibility certificate found.');

        elseif strcmp(solsta, 'MSK_SOL_STA_PRIM_INFEAS_CER')
            fprintf('Primal infeasibility certificate found.');

        elseif strcmp(solsta, 'MSK_SOL_STA_UNKNOWN')
            % The solutions status is unknown. The termination code
            % indicates why the optimizer terminated prematurely.
            fprintf('The solution status is unknown.\n');
            fprintf('Termination code: %s (%d) %s.\n', res.rcodestr, res.rcode, res.rmsg);
        else
```

(continues on next page)

```

        fprintf('An unexpected solution status is obtained.');
```

end

```

    else
        fprintf('Error during optimization: %s (%d) %s.\n', res.rcodestr, res.rcode, res.
↪rmsg);
    end

    else
        fprintf('Could not read input file, error: %s (%d) %s.\n', res.rcodestr, res.rcode, res.
↪rmsg);
    end

end
```

7.2 Errors and exceptions

Response codes

The function `mosekopt` and its variants return a **response code** (and its human-readable description), informing if optimization was performed correctly, and if not, what error occurred. The expected response, indicating successful execution, is always `"MSK_RES_OK"`. Typical errors include:

- referencing a nonexisting variable (for example with too large index),
- incompatible dimensions of input data matrices,
- NaN in the input data,
- duplicate conic variable,
- error in the optimizer.

Some errors in data preprocessing, such as incorrect command or wrong parameter value will result in `mosekopt` exiting without assigning output; the error message will just be printed out. For this reason it may be a good idea to call `mosekopt` in a try-catch block. A full list of response codes, error, warning and termination codes can be found in the [API reference](#). For example, the following code

```

prob.a = sparse(0,1);
prob.c = [NaN];
[r, res] = mosekopt('minimize', prob);
res
```

will produce as output:

```

res =

    rcode: 1470
    rmsg: 'The objective vector c contains an invalid value for variable '' (0).'
```

rcodestr: 'MSK_RES_ERR_NAN_IN_C'

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. 7.3](#)). A typical warning is, for example:

```

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

Warnings can also be suppressed by setting the `MSK_IPAR_MAX_NUM_WARNINGS` parameter to zero, if they are well-understood.

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 7.2: Sample framework for checking optimization result.

```
function response(inputfile)

cmd = sprintf('read(%s)', inputfile)
% In this example we read the problem from file
[r, res] = mosekopt(cmd)

% Read was successful
if strcmp(res.rcodestr, 'MSK_RES_OK')

    prob = res.prob;
    param = []

    % (Optionally) Uncomment the next line to get solution status Unknown
    % param.MSK_IPAR_INTPNT_MAX_ITERATIONS = 1

    % Perform the optimization.
    [r, res] = mosekopt('minimize', prob, param);

    % Expected result: The solution status of the interior-point solution is optimal.

    % Check if we have non-error termination code or OK
    if isempty(strfind(res.rcodestr, 'MSK_RES_ERR'))

        solsta = strcat('MSK_SOL_STA_', res.sol.itr.solsta)

        if strcmp(solsta, 'MSK_SOL_STA_OPTIMAL')
            fprintf('An optimal interior-point solution is located:\n');
            res.sol.itr.xx

        elseif strcmp(solsta, 'MSK_SOL_STA_DUAL_INFEAS_CER')
            fprintf('Dual infeasibility certificate found.');

        elseif strcmp(solsta, 'MSK_SOL_STA_PRIM_INFEAS_CER')
            fprintf('Primal infeasibility certificate found.');

        elseif strcmp(solsta, 'MSK_SOL_STA_UNKNOWN')
            % The solutions status is unknown. The termination code
            % indicates why the optimizer terminated prematurely.
            fprintf('The solution status is unknown.\n');
            fprintf('Termination code: %s (%d) %s.\n', res.rcodestr, res.rcode, res.rmsg);
        else
            fprintf('An unexpected solution status is obtained.');
        end

    else
        fprintf('Error during optimization: %s (%d) %s.\n', res.rcodestr, res.rcode, res.
↪rmsg);
    end

    else
        fprintf('Could not read input file, error: %s (%d) %s.\n', res.rcodestr, res.rcode, res.
↪rmsg)
```

(continues on next page)

```
end
end
```

7.3 Input/Output

7.3.1 Stream logging

By default the solver prints a log output analogous to the one produced by the command-line version of **MOSEK**. Logging may be turned off using the command `echo(0)`, for example:

```
[r, res] = mosekopt('minimize echo(0)', prob);
```

Log output may be redirected to a file using the command `log`, for example:

```
[r, res] = mosekopt('minimize log(fileName.txt)', prob);
```

Note that in recent versions of MATLAB the log is not displayed on screen until optimization is completed, which may be an inconvenience for longer tasks. The log written to a file does not have this issue.

Note also that leaving log output on can lead to a dramatic slowdown, visible especially on very small problems.

It is also possible to register a user-defined callback function that will receive and handle all log output, see the *callback* argument of *mosekopt*.

7.3.2 Log verbosity

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

- *MSK_IPAR_LOG*,
- *MSK_IPAR_LOG_INTPNT*,
- *MSK_IPAR_LOG_MIO*,
- *MSK_IPAR_LOG_CUT_SECOND_OPT*,
- *MSK_IPAR_LOG_SIM*, and
- *MSK_IPAR_LOG_SIM_MINOR*.

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

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 *MSK_IPAR_LOG*. Larger values of *MSK_IPAR_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 *MSK_IPAR_LOG_CUT_SECOND_OPT* to zero.

7.3.3 Saving a problem to a file

An optimization problem can be dumped to a file using the command `write`. The file format will be determined from the filename's extension. Supported formats are listed in [Sec. 16](#) together with a table of problem types supported by each.

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

```
[r, res] = mosekopt('write(dump.opf)', prob);
```

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

```
[r, res] = mosekopt('write(dump.task.gz)', prob);
```

When using MATLAB-like functions the file name can be set using the `options` structure, for example:

```
opt.Write = 'problem.opf';  
linprog(f,A,b,[],[],[],[],opt);
```

Some remarks:

- The problem is written to the file as it is represented in the underlying *optimizer task*, that is including any auxiliary variables introduced by the MATLAB-to-C interface, if applicable.
- 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.

7.3.4 Reading a problem from a file

A problem saved in any of the supported file formats can be read directly into a `prob` structure using the command `read`. Afterwards the problem can be optimized, modified, etc.

```
[r, res] = mosekopt('read(dump.opf.gz)');  
prob = res.prob;
```

7.4 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 name and it can accept either integers, floating point values, symbolic strings or symbolic values. Parameters are set in the structure `param` and passed as a separate argument to `mosekopt`.

Some parameters can accept symbolic strings or symbolic values from a fixed set. 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 7.3: Parameter setting example.

```
% Set log level (integer parameter)
param.MSK_IPAR_LOG = 1;
% Select interior-point optimizer... (integer parameter)
param.MSK_IPAR_OPTIMIZER = 'MSK_OPTIMIZER_INTPNT';
% ... without basis identification (integer parameter)
param.MSK_IPAR_INTPNT_BASIS = 'MSK_BI_NEVER';
% Set relative gap tolerance (double parameter)
param.MSK_DPAR_INTPNT_CO_TOL_REL_GAP = 1.0e-7;

% Use in mosekopt
[r,resp] = mosekopt('minimize', prob, param);
```

7.5 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*
- *Integer*
- *Long*

Certain information items make sense, and are made available, also *during* the optimization process. They can be accessed from a callback function, see [Sec. 7.6](#) 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 `MSK_IPAR_AUTO_UPDATE_SOL_INFO`.

Retrieving the values

Values of information items are only returned if the `info` command is used in `mosekopt`. They are available in the field `res.info`.

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 7.4: Information items example.

```
[r,res] = mosekopt('minimize info', prob);

res.info.MSK_DINF_OPTIMIZER_TIME
res.info.MSK_IINF_INTPNT_ITER
```

7.6 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.

7.6.1 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 *MSK_IPAR_LOG_SIM_FREQ* 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 callback is set by attaching a structure *callback* as a parameter in *mosekopt*. This structure specifies a global callback function and can contain arbitrary user-defined data.

7.6.2 Progress callback

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

7.6.3 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.

Listing 7.5: An example of a data callback function.

```
function [r] = callback_handler(handle,where,info)

r = 0;    % r should always be assigned a value.

if handle.symbcon.MSK_CALLBACK_BEGIN_INTPNT==where
    disp(sprintf('Interior point optimizer started\n'));
end

if handle.symbcon.MSK_CALLBACK_END_INTPNT==where
    disp(sprintf('Interior-point optimizer terminated\n'));
    disp(sprintf('Interior-point primal obj.: %e\n', info.MSK_DINF_INTPNT_PRIMAL_OBJ));
    disp(sprintf('Iterations: %d\n', info.MSK_IINF_INTPNT_ITER));
end

if handle.symbcon.MSK_CALLBACK_NEW_INT_MIO==where
    disp(sprintf('New mixed-integer solution found\n'));
    disp(sprintf('Best objective.: %e\n', info.MSK_DINF_MIO_OBJ_BOUND));
```

(continues on next page)

(continued from previous page)

```
end

% Decide if to terminate the optimization
% Terminate when cputime > handle.maxtime
if info.MSK_DINF_INTPNT_TIME > handle.maxtime
    r = 1;
else
    r = 0;
end
```

Assuming that we have defined some problem `prob` the callback function is attached as follows:

Listing 7.6: Attaching the data callback function to the model.

```
% Define user defined handle.
[r,res] = mosekopt('echo(0) symbcon');
data.maxtime = 100.0;
data.symbcon = res.symbcon;

callback.iter = 'callback_handler'; % Defined in callback_handler.m
callback.iterhandle = data;

% Perform the optimization.
[r,res] = mosekopt('minimize echo(0)',prob,[],callback);
```

Chapter 8

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:

- Consult the log output. It is enabled by default, but if necessary switch it on explicitly with:

```
[r, res] = mosekopt('minimize echo(10)', prob);
```

- Run the optimization and analyze the log output, see [Sec. 8.1](#). 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. 7.3.3](#).
 - 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.

```
[r, res] = mosekopt('write(dump.opf)', prob);
```

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

```
[r, res] = mosekopt('write(dump.task.gz)', prob);
```

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

See the following sections for details.

8.1 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](#).

8.1.1 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. 8](#)) 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. 8](#)) 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. 7.2](#) for more details.

8.1.2 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. 7.1.3](#).
- A summary of the primal solution: objective value, infinity norm of the solution vector \mathbf{xx} , maximal violations of constraints, variable bounds and cones. The violation of a linear constraint such as $a^T x \leq b$ is $\max(a^T x - b, 0)$. The violation of a conic constraint $x \in \mathcal{K}$ is the distance $\text{dist}(x, \mathcal{K})$.
- 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. 8.2](#) 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. 8.1.1](#) and [Sec. 8.3](#) 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. 8.3](#) and [Sec. 8.2](#).

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.

8.1.3 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. 8.3](#).

8.2 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.

8.2.1 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. 8.1.1](#) 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 8.3](#).

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 8.1](#).

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. 8.1.2](#) to check the objective values or solution norms.

8.2.2 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 `MSK_IPAR_OPTIMIZER` and compare the results.
- Force the optimizer to solve the primal/dual versions of the problem by setting the parameter `MSK_IPAR_INTPNT_SOLVE_FORM` or `MSK_IPAR_SIM_SOLVE_FORM`. **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 `MSK_IPAR_PRESOLVE_USE`, see [Sec. 13.1](#).
- Use different numbers of threads (`MSK_IPAR_NUM_THREADS`) 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. 8.4](#).

8.2.3 Typical pitfalls

Example 8.1 (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 8.2 (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 8.3 (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 8.4 (Presolving very tight bounds). Next consider the problem

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

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 `MSK_DPAR_PRESOLVE_TOL_X`. 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 `MSK_DPAR_PRESOLVE_TOL_X` to say 10^{-10} . This will at least make sure that presolve does not make the undesired reduction.

8.3 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.

8.3.1 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. 8.1.2](#) for how to diagnose that and [Sec. 8.2](#) for possible hints. [Sec. 8.2.3](#) contains examples of situations which may lead to infeasibility for numerical reasons.

We refer to [Sec. 8.2](#) 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.

8.3.2 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. 8.1](#).

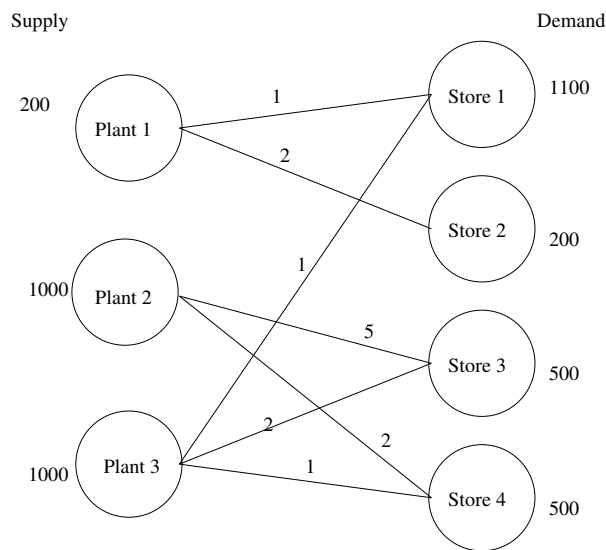


Fig. 8.1: Supply, demand and cost of transportation.

The problem represented in [Fig. 8.1](#) is infeasible, since the total demand

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

exceeds the total supply

$$2200 = 200 + 1000 + 1000$$

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

$$\begin{aligned}
 & \text{minimize} && x_{11} + 2x_{12} + 5x_{23} + 2x_{24} + x_{31} + 2x_{33} + x_{34} \\
 & \text{subject to} && s_0 : x_{11} + x_{12} \leq 200, \\
 & && s_1 : x_{23} + x_{24} \leq 1000, \\
 & && s_2 : x_{31} + x_{33} + x_{34} \leq 1000, \\
 & && d_1 : x_{11} + x_{31} = 1100, \\
 & && d_2 : x_{12} = 200, \\
 & && d_3 : x_{23} + x_{33} = 500, \\
 & && d_4 : x_{24} + x_{34} = 500, \\
 & && x_{ij} \geq 0.
 \end{aligned} \tag{8.1}$$

Solving problem (8.1) 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"`. 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.

8.3.3 Locating dual infeasibility

A problem may also be *dual infeasible*. In this case the primal problem is usually unbounded, meaning that feasible solutions exist 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 (8.1) (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					
Solution status : DUAL_INFEASIBLE_CER					
Primal. obj: 1.0000000000e+02 nrm: 1e+00 Viol. con: 0e+00 var: 0e+00					

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

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

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

8.3.4 Suggestions

Primal infeasibility

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

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

- Relax bounds of the suspicious constraints or variables.
- For integer problems, remove integrality constraints on some/all variables and see if the problem solves.
- Form an **elastic model**: allow to violate constraints at a cost. Introduce slack variables and add them to the objective as penalty. For instance, suppose we have a constraint

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

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

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

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

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

Dual infeasibility

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

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

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

solve the problem with an additional constraint such as

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

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

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

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

8.4 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. 8](#).

8.4.1 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.

8.4.2 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
```

8.4.3 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 8.1: List of commands of the MOSEK Python Console.

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

Chapter 9

Advanced Numerical Tutorials

9.1 Converting a quadratically constrained problem to conic form

MOSEK employs the following form of quadratic problems:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Q^o x + c^T x + c^f \\ & \text{subject to} && \begin{aligned} l_k^c &\leq \frac{1}{2}x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j &\leq u_k^c, & k = 0, \dots, m-1, \\ l_j^x &\leq x_j &\leq u_j^x, & j = 0, \dots, n-1. \end{aligned} \end{aligned} \quad (9.1)$$

A conic quadratic constraint has the form

$$x \in \mathcal{Q}^n$$

in its most basic form where

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

A quadratic problem such as (9.1), if convex, can be reformulated in conic form. This is in fact the reformulation **MOSEK** performs internally. It has many advantages:

- elegant duality theory for conic problems,
- reporting accurate dual information for quadratic inequalities is hard and/or computational expensive,
- it certifies that the original quadratic problem is indeed convex,
- modeling directly in conic form usually leads to a better model [And13] i.e. a faster solution time and better numerical properties.

In addition, there are more types of conic constraints that can be combined with a quadratic cone, for example semidefinite cones.

MOSEK offers a function that performs the conversion from quadratic to conic quadratic form explicitly. Note that the reformulation is not unique. The approach followed by **MOSEK** is to introduce additional variables, linear constraints and quadratic cones to obtain a larger but equivalent problem in which the original variables are preserved.

In particular:

- all variables and constraints are kept in the problem,
- each quadratic constraint and quadratic terms in the objective generate one rotated quadratic cone,
- each quadratic constraint will contain no coefficients and upper/lower bounds will be set to $\infty, -\infty$ respectively.

This allows the user to recover the original variable and constraint values, as well as their dual values, with no conversion or additional effort.

9.1.1 Quadratic Constraint Reformulation

Let us assume we want to convert the following quadratic constraint

$$l \leq \frac{1}{2}x^T Qx + \sum_{j=0}^{n-1} a_j x_j \leq u$$

to conic form. We first check whether $l = -\infty$ or $u = \infty$, otherwise either the constraint can be dropped, or the constraint is not convex. Thus let us consider the case

$$\frac{1}{2}x^T Qx + \sum_{j=0}^{n-1} a_j^T x_j \leq u. \quad (9.2)$$

Introducing an additional variable w such that

$$w = u - \sum_{j=0}^{n-1} a_j^T x_j \quad (9.3)$$

we obtain the equivalent form

$$\begin{aligned} \frac{1}{2}x^T Qx &\leq w, \\ u - \sum_{j=0}^{n-1} a_j^T x_j &= w. \end{aligned}$$

If Q is positive semidefinite, then there exists a matrix F such that

$$Q = FF^T \quad (9.4)$$

and therefore we can write

$$\begin{aligned} \|Fx\|^2 &\leq 2w, \\ u - \sum_{j=0}^{n-1} a_j^T x_j &= w. \end{aligned}$$

Introducing an additional variable $z = 1$, and setting $y = Fx$ we obtain the conic formulation

$$\begin{aligned} (w, z, y) &\in \mathcal{Q}_r, \\ z &= 1 \\ y &= Fx \\ w &= u - a^T x. \end{aligned} \quad (9.5)$$

Summarizing, for each quadratic constraint involving t variables, **MOSEK** introduces

1. a rotated quadratic cone of dimension $t + 2$,
2. two additional variables for the cone roots,
3. t additional variables to map the remaining part of the cone,
4. t linear constraints.

A quadratic term in the objective is reformulated in a similar fashion. We refer to [\[And13\]](#) for a more thorough discussion.

Example

Next we consider a simple problem with quadratic objective function:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}(13x_0^2 + 17x_1^2 + 12x_2^2 + 24x_0x_1 + 12x_1x_2 - 4x_0x_2) - 22x_0 - 14.5x_1 + 12x_2 + 1 \\ \text{subject to} \quad & -1 \leq x_0, x_1, x_2 \leq 1 \end{aligned}$$

We can specify it in the human-readable OPF format.

```

[comment]
An example of small QO problem from Boyd and Vandenberghe, "Convex Optimization", page 189 ex.
↪4.3
The solution is (1,0.5,-1)
[/comment]

[variables]
x0 x1 x2
[/variables]

[objective min]
0.5 (13 x0^2 + 17 x1^2 + 12 x2^2 + 24 x0 * x1 + 12 x1 * x2 - 4 x0 * x2 ) - 22 x0 - 14.5 x1 +
↪12 x2 + 1
[/objective]

[bounds]
[b] -1 <= * <= 1 [/b]
[/bounds]

```

The objective function is convex, the minimum is attained for $x^* = (1, 0.5, -1)$. The conversion will introduce first a variable x_3 in the objective function such that $x_3 \geq 1/2x^T Qx$ and then convert the latter directly in conic form. The converted problem follows:

$$\begin{aligned}
&\text{minimize} && -22x_0 - 14.5x_1 + 12x_2 + x_3 + 1 \\
&\text{subject to} && 3.61x_0 + 3.33x_1 - 0.55x_2 - x_6 = 0 \\
& && +2.29x_1 + 3.42x_2 - x_7 = 0 \\
& && 0.81x_1 - x_8 = 0 \\
& && -x_3 + x_4 = 0 \\
& && x_5 = 1 \\
& && (x_4, x_5, x_6, x_7, x_8) \in \mathcal{Q}_\nabla \\
& && -1 \leq x_0, x_1, x_2 \leq 1
\end{aligned}$$

We obtain the reformulation as follows:

```

% prob is a quadratic problem
[r, res] = mosekopt('toconic prob', prob)
probConic = res.prob
mosekopt('write(conic.opf)', probConic)

```

and the output is:

```

[comment]
  Written by MOSEK version 8.1.0.19
  Date 21-08-17
  Time 10:53:36
[/comment]

[hints]
[hint NUMVAR] 9 [/hint]
[hint NUMCON] 4 [/hint]
[hint NUMANZ] 11 [/hint]
[hint NUMQNZ] 0 [/hint]
[hint NUMCONE] 1 [/hint]
[/hints]

[variables disallow_new_variables]
x0000_x0 x0001_x1 x0002_x2 x0003_x0004
x0005_x0006 x0007_x0008
[/variables]

[objective minimize]
- 2.2e+01 x0000_x0 - 1.45e+01 x0001_x1 + 1.2e+01 x0002_x2 + x0003
+ 1e+00

```

(continues on next page)

```
[/objective]

[constraints]
[con c0000] 3.605551275463989e+00 x0000_x0 - 5.547001962252291e-01 x0002_x2 + 3.
↪328201177351375e+00 x0001_x1 - x0006 = 0e+00 [/con]
[con c0001] 3.419401657060442e+00 x0002_x2 + 2.294598480395823e+00 x0001_x1 - x0007 = 0e+00 ↪
↪[/con]
[con c0002] 8.111071056538127e-01 x0001_x1 - x0008 = 0e+00 [/con]
[con c0003] - x0003 + x0004 = 0e+00 [/con]
[/constraints]

[bounds]
[b] -1e+00 <= x0000_x0,x0001_x1,x0002_x2 <= 1e+00 [/b]
[b] x0003,x0004 free [/b]
[b] x0005 = 1e+00 [/b]
[b] x0006,x0007,x0008 free [/b]
[cone rquad k0000] x0004, x0005, x0006, x0007, x0008 [/cone]
[/bounds]
```

We can clearly see that constraints c0000, c0001 and c0002 represent the original linear constraints as in (9.4), while c0003 corresponds to (9.3). The cone roots are x0005 and x0004.

9.2 Advanced hot-start

In practice it frequently occurs that when an optimization problem has been solved, then the same problem slightly modified should be reoptimized. Moreover, if it is just a small the modification, it can be expected that the optimal solution to the original problem is a good approximation to the modified problem. Therefore, it should be efficient to start the optimization of the modified problem from the previous optimal solution.

Currently, the interior-point optimizer in **MOSEK** cannot take advantage of a previous optimal solution, however, the simplex optimizer can exploit any basic solution.

We work with the simple linear problem:

$$\begin{aligned} &\text{minimize} && x_1 + 2x_2 \\ &\text{subject to} && 4 \leq x_1 + x_3 \leq 6, \\ & && 1 \leq x_1 + x_2, \\ & && 0 \leq x_1, x_2, x_3. \end{aligned}$$

9.2.1 Initial hot-start

A quick inspection of the problem indicates that $(x_1, x_3) = (1, 3)$ is an optimal solution. Hence, it seems to be a good idea to let the initial basis consist of x_1 and x_3 and all the other variables be at their lower bounds. This idea is used in the example code:

Listing 9.1: Passing the full basic solution.

```
% Specify an initial basic solution.
bas.skc = ['LL'; 'LL'];
bas.skx = ['BS'; 'LL'; 'BS'];
bas.xc = [4 1]';
bas.xx = [1 3 0]';

prob.sol.bas = bas;

% Specify the problem data.
prob.c = [ 1 2 0]';
subi = [1 2 2 1];
subj = [1 1 2 3];
valij = [1.0 1.0 1.0 1.0];
```

(continues on next page)

(continued from previous page)

```
prob.a      = sparse(subi,subj,valij);
prob.blc    = [4.0 1.0]';
prob.buc    = [6.0 inf]';
prob.blx    = sparse(3,1);
prob.bux    = [];

% Use the primal simplex optimizer.
param.MSK_IPAR_OPTIMIZER = 'MSK_OPTIMIZER_PRIMAL_SIMPLEX';
[r,res] = mosekopt('minimize',prob,param)
```

Comments:

- In the example the dual solution is not defined. This is acceptable because the primal simplex optimizer is used for the reoptimization and it does not exploit a dual solution. Otherwise it will be important that a good dual solution is specified.
- The status keys `bas.skc` and `bas.sku` must contain only the entries BS, EQ, LL, UL, SB. Moreover, e.g. EQ must be specified only for a fixed constraint or variable. LL and UL can be used only for a variable that has a finite lower or upper bound respectively. For an explanation of status keys see *stakey*.
- The number of constraints and variables defined to be basic must correspond exactly to the number of constraints.

9.2.2 Adding a new variable

Next, assume we modify the problem by adding a new variable:

$$\begin{aligned} \text{minimize} \quad & x_1 + 2x_2 - x_4 \\ \text{subject to} \quad & 4 \leq x_1 + x_3 + x_4 \leq 6, \\ & 1 \leq x_1 + x_2, \\ & 0 \leq x_1, x_2, x_3, x_4. \end{aligned}$$

In continuation of the previous example this problem can be solved as follows, using the full previous basic solution in hot-start:

Listing 9.2: Hot-start when adding a new variable.

```
prob.c      = [prob.c;-1.0];
prob.a      = [prob.a,sparse([1.0 0.0]')];
prob.blx    = sparse(4,1);

% Reuse the old optimal basic solution.
bas         = res.sol.bas;

% Add to the status key.
bas.sku     = [res.sol.bas.sku;'LL'];

% The new variable is at it lower bound.
bas.xx      = [res.sol.bas.xx;0.0];
bas.slx     = [res.sol.bas.slx;0.0];
bas.sux     = [res.sol.bas.sux;0.0];

prob.sol.bas = bas;

[rcode,res] = mosekopt('minimize',prob,param);

% The new primal optimal solution
res.sol.bas.xx'
```

9.2.3 Fixing a variable

In e.g. branch-and-bound methods for integer programming problems it is necessary to reoptimize the problem after a variable has been fixed to a value. This can easily be achieved as follows:

Listing 9.3: Hot-start with a fixed variable.

```
prob.blx(4) = 1;
prob.bux    = [inf inf inf 1]';

% Reuse the basis.
prob.sol.bas = res.sol.bas;

[rcode,res] = mosekopt('minimize',prob,param);

% Display the optimal solution.
res.sol.bas.xx'
```

9.2.4 Adding a new constraint

Now assume that the constraint

$$x_1 + x_2 \geq 2$$

should be added to the problem and the problem should be reoptimized. The following example demonstrates how to do this.

Listing 9.4: Hot-start when adding a new constraint.

```
% Modify the problem.
prob.a      = [prob.a;sparse([1.0 1.0 0.0 0.0])];
prob.blc    = [prob.blc;2.0];
prob.buc    = [prob.buc;inf];

% Obtain the previous optimal basis.
bas         = res.sol.bas;

% Set the solution to the modified problem.
bas.skc     = [bas.skc;'BS'];
bas.xc      = [bas.xc;bas.xx(1)+bas.xx(2)];
bas.y       = [bas.y;0.0];
bas.slc     = [bas.slc;0.0];
bas.suc     = [bas.suc;0.0];

% Reuse the basis.
prob.sol.bas = bas;

% Reoptimize.
[rcode,res] = mosekopt('minimize',prob,param);

res.sol.bas.xx'
```

Please note that the slack variable corresponding to the new constraint is declared basic. This implies that the new basis is nonsingular and can be reused.

9.2.5 Removing a constraint

We can remove a constraint in two ways:

- Set the bounds for the constraint to $\pm\infty$ as appropriate.
- Remove the corresponding row from `prob.a` and other parts of the data and update the basis.

In the following example we use the latter approach to again remove the constraint $x_1 + x_2 \geq 2$.

Listing 9.5: Hot-start when removing a constraint.

```
% Modify the problem.
prob.a      = prob.a(1:end-1,:);
prob.blc    = prob.blc(1:end-1);
prob.buc    = prob.buc(1:end-1);

% Obtain the previous optimal basis.
bas         = res.sol.bas;

% Set the solution to the modified problem.
bas.skc     = bas.skc(1:end-1,:);
bas.xc      = bas.xc(1:end-1);
bas.y       = bas.y(1:end-1);
bas.slc     = bas.slc(1:end-1);
bas.suc     = bas.suc(1:end-1);

% Reuse the basis.
prob.sol.bas = bas;

% Reoptimize.
[rcode,res] = mosekopt('minimize',prob,param);

res.sol.bas.xx'
```

9.2.6 Removing a variable

Similarly we can remove a variable in two ways:

- Fix the variable to zero.
- Remove the corresponding column from `prob.a` and other parts of the data and update the basis.

The following example uses the latter approach to remove x_4 .

Listing 9.6: Hot-start when removing a constraint.

```
% Modify the problem.
prob.c      = prob.c(1:end-1);
prob.a      = prob.a(:,1:end-1);
prob.blx    = prob.blx(1:end-1);
prob.bux    = prob.bux(1:end-1);

% Obtain the previous optimal basis.
bas         = res.sol.bas;

% Set the solution to the modified problem.
bas.xx      = bas.xx(1:end-1);
bas.skx     = bas.skx(1:end-1,:);
bas.slx     = bas.slx(1:end-1);
bas.sux     = bas.sux(1:end-1);

% Reuse the basis.
prob.sol.bas = bas;

% Reoptimize.
[rcode,res] = mosekopt('minimize',prob,param);

res.sol.bas.xx'
```

Chapter 10

Technical guidelines

This section contains some more in-depth technical guidelines for Optimization Toolbox for MATLAB, not strictly necessary for basic use of **MOSEK**.

10.1 Integration with MATLAB

The `mosekopt` MEX file

The central part of Optimization Toolbox for MATLAB is the `mosekopt` MEX file. It provides an interface to **MOSEK** that is employed by all the other functions provided in the toolbox. Therefore, we recommend to `mosekopt` function if possible because that give rise to the least overhead and provides the maximum of features.

Compatibility with the MATLAB Optimization Toolbox

For compatibility with the MATLAB Optimization Toolbox, **MOSEK** provides the following functions:

- `linprog`: Solves linear optimization problems.
- `intlinprog`: Solves a linear optimization problem with integer constrained variables.
- `quadprog`: Solves quadratic optimization problems.
- `lsqlin`: Minimizes a least-squares objective with linear constraints.
- `lsqnonneg`: Minimizes a least-squares objective with nonnegativity constraints.
- `mskoptimget`: Getting an `options` structure for MATLAB compatible functions.
- `mskoptimset`: Setting up an `options` structure for MATLAB compatible functions.

These functions are described in detail in [Sec. 15.2](#). The functions `mskoptimget` and `mskoptimset` are not fully compatible with the MATLAB counterparts, `optimget` and `optimset`, so the **MOSEK** versions should only be used in conjunction with the **MOSEK** implementations of `linprog`, etc., and similarly `optimget` should be used in conjunction with the MATLAB implementations.

Caveats using the MATLAB compiler

When using **MOSEK** with the MATLAB compiler it is necessary manually:

- to remove `mosekopt.m` before compilation,
- copy the MEX file to the directory with MATLAB binary files and
- copy the `mosekopt.m` file back after compilation.

10.2 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.

Names of various elements of the problem are assigned using the *names* structure within an optimization problem specification *prob*.

10.3 Multithreading

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 *MSK_IPAR_NUM_THREADS* 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 *MSK_IPAR_NUM_THREADS*.

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 *MSK_IPAR_NUM_THREADS* 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 *MSK_IPAR_INTPNT_MULTI_THREAD*.

The MATLAB Parallel Computing Toolbox

Running **MOSEK** with the MATLAB Parallel Computing Toolbox requires multiple **MOSEK** licenses, since each thread runs a separate instance of the **MOSEK** optimizer. Each thread thus requires a **MOSEK** license.

10.4 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 any **MOSEK** function involving optimization, as for instance *mosekopt*, is called the first time and
- it is returned when MATLAB is terminated.

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 *MSK_IPAR_CACHE_LICENSE* to "*MSK_OFF*" will force **MOSEK** to return the license token immediately after the optimization completed.
- Setting the parameter *MSK_IPAR_LICENSE_WAIT* will force **MOSEK** to wait until a license token becomes available instead of returning with an error.
- All licenses currently checked out and not in use can be released on demand using the **nokeepenv** command of *mosekopt*.

```
mosekopt('nokeepenv');
```

Chapter 11

Case Studies

In this section we present some case studies in which the Optimization Toolbox for MATLAB 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. 6](#) 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
- *Least squares and other norm minimization problems*
 - **Keywords:** Least squares, regression, 2-norm, 1-norm, p-norm, ridge, lasso
 - **Type:** Conic Quadratic, Power Cone
- *Robust linear optimization*
 - **Keywords:** Robust optimization, ellipsoidal uncertainty
 - **Type:** Conic Quadratic

11.1 Portfolio Optimization

In this section the Markowitz portfolio optimization problem and variants are implemented using Optimization Toolbox for MATLAB.

- *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

$$\Sigma = GG^T. \quad (11.2)$$

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

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

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

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

which is a conic quadratic optimization problem that can easily be formulated and solved with Optimization Toolbox for MATLAB. 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).

```
function er = BasicMarkowitz(n,mu,GT,x0,w,gamma)

[rcode, res] = mosekopt('symbcon');
prob = [];

% Objective vector - expected return
prob.c = mu;

% The budget constraint - e'x = w + sum(x0)
prob.a = ones(1,n);
prob.blc = w + sum(x0);
prob.buc = w + sum(x0);

% Bounds exclude shortselling
prob.blx = zeros(n,1);
prob.bux = inf*ones(n,1);

% An affine conic constraint: [gamma, GT*x] in quadratic cone
prob.f = sparse([ zeros(1,n); GT ]);
prob.g = [gamma; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD n+1 ];

% Maximize problem and return the objective value
[rcode,res] = mosekopt('maximize echo(0)', prob, []);
x = res.sol.itr.xx;
er = mu'*x;
```

The source code should be self-explanatory except perhaps for

```
prob.f = sparse([ zeros(1,n); GT ]);
prob.g = [gamma; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD n+1 ];
```

where the constraint

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

is created as an *affine conic constraint format* of the form $Fx + g \in \mathcal{K}$, in this specific case:

$$\begin{bmatrix} 0 \\ G^T \end{bmatrix} x + \begin{bmatrix} \gamma \\ 0 \end{bmatrix} \in \mathcal{Q}^{n+1}.$$

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.

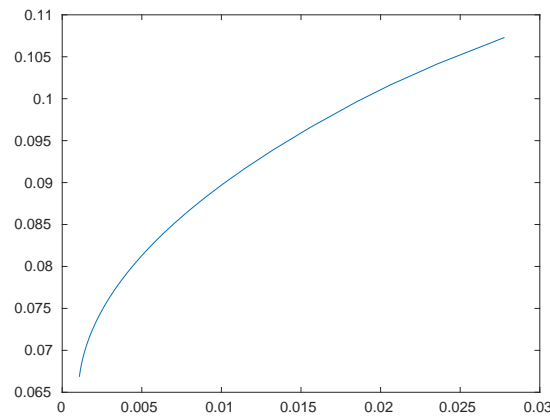


Fig. 11.1: The efficient frontier for the sample data.

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).

```
function frontier = EfficientFrontier(n,mu,GT,x0,w,alphas)

frontier = [];
[rcode, res] = mosekopt('symbcon');
prob = [];

% The budget constraint in terms of variables [x; s]
prob.a = [ones(1,n), 0.0];
prob.blc = w + sum(x0);
prob.buc = w + sum(x0);

% No shortselling
prob.blx = [zeros(n,1); -inf];
prob.bux = inf*ones(n+1,1);

% An affine conic constraint: [s, 0.5, GT*x] in rotated quadratic cone
% In matrix form
% [ 0 1] [ x ]      [ 0 ]
% [ 0 0] [ ] + [ 0.5 ] \in Q_r
% [ GT 0] [ s ]      [ 0 ]
prob.f = sparse([ zeros(1,n), 1.0]; zeros(1, n+1); [GT, zeros(n,1)] );
prob.g = [ 0; 0.5; zeros(n, 1) ]
prob.cones = [ res.symbcon.MSK_CT_RQUAD n+2 ];

for alpha = alphas
    % Objective mu'*x - alpha*s
    prob.c = [mu; -alpha];

    [rcode,res] = mosekopt('maximize echo(0)',prob,[]);
    x = res.sol.itr.xx(1:n);
    s = res.sol.itr.xx(n+1);

    frontier = [frontier; [alpha, mu'*x, s] ];
end
```

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

$$z_j > |x_j - x_j^0| \quad (11.9)$$

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

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

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).

```
function [x, t] = MarkowitzWithMarketImpact(n,mu,GT,x0,w,gamma,m)

[rcode, res] = mosekopt('symcon');

% unrolled variable ordered as (x, t)
prob = [];
prob.c = [mu; zeros(n,1)];

In = speye(n);
On = sparse([], [], [], n, n);

% Linear part
% [ e' m' ] * [ x; t ] = w + e'*x0
prob.a = [ ones(1,n), m' ];
prob.blc = [ w + sum(x0) ];
prob.buc = [ w + sum(x0) ];

% No shortselling and no other bounds
prob.blx = [ zeros(n,1); -inf*ones(n,1) ];
prob.bux = [ inf*ones(2*n,1) ];

% Affine conic constraints representing [ gamma, GT*x ] in quadratic cone
```

(continues on next page)

(continued from previous page)

```

prob.f = sparse([ zeros(1,2*n); [GT On] ]);
prob.g = [gamma; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD n+1 ];

% Extend the affine conic constraints
% with power cones representing t(i) >= |x(i)-x0(i)|^1.5
fi = [];
fj = [];
g = [];
fv = repmat([1; 1], n, 1);
for k=1:n
    fi = [fi; 3*k-2; 3*k];
    fj = [fj; n+k; k];
    g = [g; 0; 1; -x0(k)];
end
prob.f = [prob.f; sparse(fi, fj, fv)];
prob.g = [prob.g; g];
prob.cones = [prob.cones repmat([res.symbcon.MSK_CT_PPOW, 3, 2, 2.0, 1.0], 1, n) ];

[rcode,res] = mosekopt('maximize echo(0)',prob,[]);

x = res.sol.itr.xx(1:n);
t = res.sol.itr.xx(n+(1:n));

```

In the last part of the code we extend the affine conic constraint with triples of the form $(t_k, 1, x_k - x_k^0)$. Such a triple is constructed as an affine conic constraint with:

$$\begin{bmatrix} e_{n+k}^T \\ 0 \\ e_k^T \end{bmatrix} \cdot \begin{bmatrix} x \\ t \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -x_k^0 \end{bmatrix}$$

where e_j denotes the vector of length $2n$ with 1 at position j and 0 otherwise. Membership of a sequence of triples in power cones $\mathcal{P}_3^{2/3, 1/3}$ is specified with the syntax:

```
prob.cones = [prob.cones repmat([res.symbcon.MSK_CT_PPOW, 3, 2, 2.0, 1.0], 1, n) ];
```

Note that the construction `[res.symbcon.MSK_CT_PPOW, d, 2, a, b]` creates a power cone of dimension d with exponents

$$\frac{a}{a+b}, \frac{b}{a+b}.$$

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

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

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).

```
function [x, z, y] = MarkowitzWithTransactionsCost(n,mu,GT,x0,w,gamma,f,g)

[rcline, res] = mosekopt('symbcon');

% Upper bound on the traded amount
u = w+sum(x0);

% unrolled variable ordered as (x, z, y)
prob = [];
prob.c = [mu; zeros(2*n,1)];
In = speye(n);
On = sparse([],[],[],n,n);

% Linear constraints
% [ e'  g'  f' ] [ x ] = w + e'*x0
% [ I  -I  0 ] * [ z ] <= x0
% [ I  I  0 ] [ y ] >= x0
% [ 0  I  -U ] <= 0
prob.a = [ ones(1,n), g', f']; In -In On; In In On; On In -u*In ];
prob.blc = [ w + sum(x0); -inf*ones(n,1); x0; -inf*ones(n,1) ];
prob.buc = [ w + sum(x0); x0; inf*ones(n,1); zeros(n,1) ];

% No shortselling and the linear bound 0 <= y <= 1
prob.blx = [ zeros(n,1); -inf*ones(n,1); zeros(n,1) ];
prob.bux = [ inf*ones(2*n,1); ones(n,1) ];

% Affine conic constraints representing [ gamma, GT*x ] in quadratic cone
prob.f = sparse([ zeros(1,3*n); [GT On On]; ]);
prob.g = [gamma; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD n+1 ];

% Demand y to be integer (hence binary)
prob.ints.sub = 2*n+(1:n);

[rcline,res] = mosekopt('maximize echo(0)',prob,[]);

x = res.sol.int.xx(1:n);
z = res.sol.int.xx(n+(1:n));
y = res.sol.int.xx(2*n+(1:n));
```

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:

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

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.

Listing 11.5: Code solving problem (11.12).

```

function x = MarkowitzWithCardinality(n,mu,GT,x0,w,gamma,k)

[rcode, res] = mosekopt('symbcon');

% Upper bound on the traded amount
u = w+sum(x0);

% unrolled variable ordered as (x, z, y)
prob = [];
prob.c = [mu; zeros(2*n,1)];
In = speye(n);
On = sparse([],[],[],n,n);

% Linear constraints
% [ e'  0  0 ]      = w + e'*x0
% [ I  -I  0 ] [ x ] <= x0
% [ I   I  0 ] * [ z ] >= x0
% [ 0   I -U ] [ y ] <= 0
% [ 0   0  e' ]      <= k
prob.a = [ [ones(1,n), zeros(1,2*n)]; In -In On; In In On; On In -u*In; zeros(1,2*n) ones(1,
↪n) ];
prob.blc = [ w + sum(x0); -inf*ones(n,1); x0; -inf*ones(n,1); 0 ];
prob.buc = [ w + sum(x0); x0; inf*ones(n,1); zeros(n,1); k ];

% No shortselling and the linear bound 0 <= y <= 1
prob.blx = [ zeros(n,1); -inf*ones(n,1); zeros(n,1) ];
prob.bux = [ inf*ones(2*n,1); ones(n,1) ];

% Affine conic constraints representing [ gamma, GT*x ] in quadratic cone
prob.f = sparse([ zeros(1,3*n); [GT On On]; ]);
prob.g = [gamma; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD n+1 ];

% Demand y to be integer (hence binary)
prob.ints.sub = 2*n+(1:n);

[rcode,res] = mosekopt('maximize echo(0)',prob,[]);

x = res.sol.int.xx(1:n);

```

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 Least Squares and Other Norm Minimization Problems

A frequently occurring problem in statistics and in many other areas of science is a norm minimization problem

$$\begin{aligned} & \text{minimize} && \|Fx - g\|, \\ & \text{subject to} && Ax = b, \end{aligned} \tag{11.13}$$

where $x \in \mathbb{R}^n$ and of course we can allow other types of constraints. The objective can involve various norms: infinity norm, 1-norm, 2-norm, p -norms and so on. For instance the most popular case of the 2-norm corresponds to the least squares linear regression, since it is equivalent to minimization of $\|Fx - g\|_2^2$.

11.2.1 Least squares, 2-norm

In the case of the 2-norm we specify the problem directly in conic quadratic form

$$\begin{aligned} & \text{minimize} && t, \\ & \text{subject to} && (t, Fx - g) \in \mathcal{Q}^{k+1}, \\ & && Ax = b. \end{aligned} \tag{11.14}$$

The first constraint of the problem can be represented as an affine conic constraint. This leads to the following model.

Listing 11.6: Script solving problem (11.14)

```
% Least squares regression
% minimize \|Fx-g\|_2
function x = norm_lse(F,g,A,b)
clear prob;
[r, res] = mosekopt('symbcon');
n = size(F,2);
k = size(g,1);
m = size(A,1);

% Linear constraints in [x; t]
prob.a = [A, zeros(m,1)];
prob.buc = b;
prob.blc = b;
prob.blx = -inf*ones(n+1,1);
prob.bux = inf*ones(n+1,1);
prob.c = [zeros(n,1); 1];

% Affine conic constraint
prob.f = sparse([zeros(1,n), 1; F, zeros(k,1)]);
prob.g = [0; -g];
prob.cones = [ res.symbcon.MSK_CT_QUAD k+1 ];

% Solve
[r, res] = mosekopt('minimize echo(0)', prob);
x = res.sol.itr.xx(1:n);
end
```

11.2.2 Ridge regularisation

Regularisation is classically applied to reduce the impact of outliers and to control overfitting. In the conic version of *ridge* (Tychonov) *regression* we consider the problem

$$\begin{aligned} & \text{minimize} && \|Fx - g\|_2 + \gamma\|x\|_2, \\ & \text{subject to} && Ax = b, \end{aligned} \quad (11.15)$$

which can be written explicitly as

$$\begin{aligned} & \text{minimize} && t_1 + \gamma t_2, \\ & \text{subject to} && (t_1, Fx - g) \in \mathcal{Q}^{k+1}, \\ & && (t_2, x) \in \mathcal{Q}^{n+1}, \\ & && Ax = b. \end{aligned} \quad (11.16)$$

The implementation is a small extension of that from the previous section.

Listing 11.7: Script solving problem (11.16)

```
% Least squares regression with regularization
% minimize ||Fx-g||_2 + gamma*||x||_2
function x = norm_lse_reg(F,g,A,b,gamma)
clear prob;
[r, res] = mosekopt('symcon');
n = size(F,2);
k = size(g,1);
m = size(A,1);

% Linear constraints in [x; t1; t2]
prob.a = [A, zeros(m,2)];
prob.buc = b;
prob.blc = b;
prob.blx = -inf*ones(n+2,1);
prob.bux = inf*ones(n+2,1);
prob.c = [zeros(n,1); 1; gamma];

% Affine conic constraint
prob.f = sparse([zeros(1,n), 1, 0; ...
                F, zeros(k,2); ...
                zeros(1,n), 0, 1; ...
                eye(n), zeros(n,2)]);
prob.g = [0; -g; zeros(n+1,1)];
prob.cones = [ res.symbcon.MSK_CT_QUAD k+1 res.symbcon.MSK_CT_QUAD n+1 ];

% Solve
[r, res] = mosekopt('minimize echo(0)', prob);
x = res.sol.itr.xx(1:n);
end
```

Note that classically least squares problems are formulated as quadratic problems and then the objective function would be written as

$$\|Fx - g\|_2^2 + \gamma\|x\|_2^2.$$

This version can easily be obtained by replacing the quadratic cone with an appropriate rotated quadratic cone in (11.16). Then the core of the implementation would change as follows:

Listing 11.8: Script solving classical quadratic ridge regression

```
prob.f = sparse([zeros(1,n), 1, 0; ...
                zeros(1,n+2) ; ...
                F, zeros(k,2); ...
                zeros(1,n), 0, 1; ...
```

(continues on next page)

(continued from previous page)

```
zeros(1,n+2)           ; ...
eye(n),      zeros(n,2) ]);
prob.g = [0; 0.5; -g; 0; 0.5; zeros(n,1)];
prob.cones = [ res.symbcon.MSK_CT_RQUAD k+2 res.symbcon.MSK_CT_RQUAD n+2 ];
```

Fig. 11.2 shows the solution to a polynomial fitting problem for a few variants of least squares regression with and without ridge regularization.

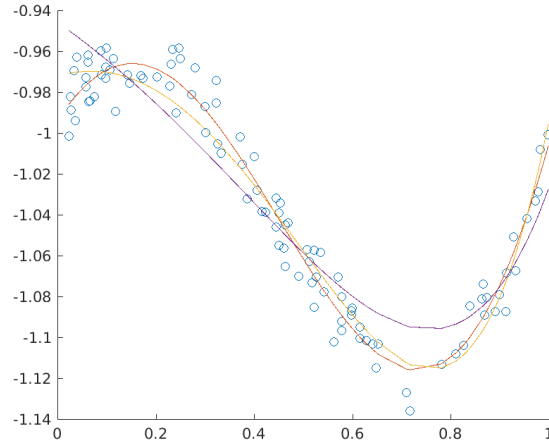


Fig. 11.2: Three fits to a dataset at various levels of regularization.

11.2.3 Lasso regularization

In *lasso* or *least absolute shrinkage and selection operator* the regularization term is the 1-norm of the solution

$$\begin{aligned} & \text{minimize} && \|Fx - g\|_2 + \gamma\|x\|_1, \\ & \text{subject to} && Ax = b. \end{aligned} \quad (11.17)$$

This variant typically tends to give preference to sparser solutions, i.e. solutions where only a few elements of x are nonzero, and therefore it is used as an efficient approximation to the cardinality constrained problem with an upper bound on the 0-norm of x . To see how it works we first implement (11.17) adding the constraint $t \geq \|x\|_1$ as a series of linear constraints

$$u_i \geq -x_i, \quad u_i \geq x_i, \quad t \geq \sum u_i,$$

so that eventually the problem becomes

$$\begin{aligned} & \text{minimize} && t_1 + \gamma t_2, \\ & \text{subject to} && u + x \geq 0, \\ & && u - x \geq 0, \\ & && t_2 - e^T u \geq 0, \\ & && Ax = b, \\ & && (t_1, Fx - g) \in \mathcal{Q}^{k+1}. \end{aligned}$$

Listing 11.9: Script solving problem (11.17)

```
% Least squares regression with lasso regularization
% minimize ||Fx-g||_2 + gamma*||x||_1
function x = norm_lse_lasso(F,g,A,b,gamma)
clear prob;
```

(continues on next page)

```

[r, res] = mosekopt('symbcon');
n = size(F,2);
k = size(g,1);
m = size(A,1);

% Linear constraints in [x; u; t1; t2]
prob.a = [A,          zeros(m,n+2)      ; ...
          eye(n),     eye(n), zeros(n,2); ...
          -eye(n),    eye(n), zeros(n,2); ...
          zeros(1,n)  -ones(1,n), 0, 1 ];
prob.buc = [b; inf*ones(2*n+1,1)];
prob.blc = [b; zeros(2*n+1,1)];
prob.blx = -inf*ones(2*n+2,1);
prob.bux = inf*ones(2*n+2,1);
prob.c = [zeros(2*n,1); 1; gamma];

% Affine conic constraint
prob.f = sparse([zeros(1,2*n), 1, 0; F, zeros(k,n+2)]);
prob.g = [0; -g];
prob.cones = [ res.symbcon.MSK_CT_QUAD k+1 ];

% Solve
[r, res] = mosekopt('minimize echo(0)', prob);
x = res.sol.itr.xx(1:n);
end

```

The sparsity pattern of the solution of a large random regression problem can look for example as follows:

Lasso regularization			
Gamma 0.0100	density 99%	Fx-g ₂ :	54.3722
Gamma 0.1000	density 87%	Fx-g ₂ :	54.3939
Gamma 0.3000	density 67%	Fx-g ₂ :	54.5319
Gamma 0.6000	density 40%	Fx-g ₂ :	54.8379
Gamma 0.9000	density 26%	Fx-g ₂ :	55.0720
Gamma 1.3000	density 12%	Fx-g ₂ :	55.1903

11.2.4 p-norm minimization

Now we consider the minimization of the p -norm defined for $p > 1$ as

$$\|y\|_p = \left(\sum_i |y_i|^p \right)^{1/p}. \quad (11.18)$$

We have the optimization problem:

$$\begin{aligned} & \text{minimize} && \|Fx - g\|_p, \\ & \text{subject to} && Ax = b. \end{aligned} \quad (11.19)$$

Increasing the value of p forces stronger penalization of outliers as ultimately, when $p \rightarrow \infty$, the p -norm $\|y\|_p$ converges to the infinity norm $\|y\|_\infty$ of y . According to the [Modeling Cookbook](#) the p -norm bound $t \geq \|Fx - g\|_p$ can be added to the model using a sequence of three-dimensional power cones and we obtain an equivalent problem

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && (r_i, t, (Fx - g)_i) \in \mathcal{P}_3^{1/p, 1-1/p}, \\ & && e^T r = t, \\ & && Ax = b. \end{aligned} \quad (11.20)$$

The power cones can be added one by one to the structure representing affine conic constraints. Each power cone will require one r_i , one copy of t and one row from F and g . An alternative solution is to

create the vector

$$[r_1; \dots; r_k; t; \dots; t; Fx - g]$$

and then reshuffle its elements into

$$[r_1; t; F_1x - g_1; \dots; r_k; t; F_kx - g_k]$$

using an appropriate permutation matrix. This approach is demonstrated in the code below.

Listing 11.10: Script solving problem (11.20)

```
% P-norm minimization
% minimize ||Fx-g||_p
function x = norm_p_norm(F,g,A,b,p)
clear prob;
[r, res] = mosekopt('symbcon');
n = size(F,2);
k = size(g,1);
m = size(A,1);

% Linear constraints in [x; r; t]
prob.a = [A, zeros(m,k+1); zeros(1,n), ones(1,k), -1];
prob.buc = [b; 0];
prob.blc = [b; 0];
prob.blx = -inf*ones(n+k+1,1);
prob.bux = inf*ones(n+k+1,1);
prob.c = [zeros(n+k,1); 1];

% Permutation matrix which picks triples (r_i, t, F_i x - g_i)
M = [];
for i=1:3
    M = [M, sparse(i:3:3*k, 1:k, ones(k,1), 3*k, k)];
end

% Affine conic constraint
prob.f = M * sparse([zeros(k,n), eye(k), zeros(k,1); zeros(k,n+k), ones(k,1); F, zeros(k,
    k+1)]);
prob.g = M * [zeros(2*k,1); -g];
prob.cones = [ repmat([res.symbcon.MSK_CT_PPOW, 3, 2, 1.0, p-1], 1, k) ];

% Solve
[r, res] = mosekopt('minimize echo(0)', prob);
x = res.sol.itr.xx(1:n);
end
```

11.3 Robust linear Optimization

In most linear optimization examples discussed in this manual it is implicitly assumed that the problem data, such as c and A , is known with certainty. However, in practice this is seldom the case, e.g. the data may just be roughly estimated, affected by measurement errors or be affected by random events.

In this section a robust linear optimization methodology is presented which removes the assumption that the problem data is known exactly. Rather it is assumed that the data belongs to some set, i.e. a box or an ellipsoid.

The computations are performed using the **MOSEK** optimization toolbox for MATLAB but could equally well have been implemented using the **MOSEK** API.

This section is co-authored with A. Ben-Tal and A. Nemirovski. For further information about robust linear optimization consult [BTN00], [BenTalN01].

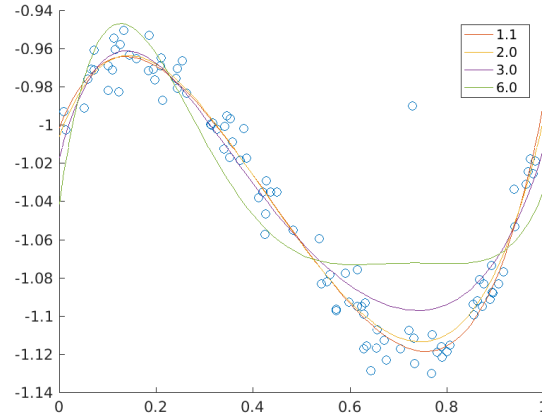


Fig. 11.3: p -norm minimizing fits of a polynomial of degree at most 5 to the data for various values of p .

11.3.1 Introductory Example

Consider the following toy-sized linear optimization problem: A company produces two kinds of drugs, DrugI and DrugII, containing a specific active agent A, which is extracted from a raw materials that should be purchased on the market. The drug production data are as follows:

Selling price \$ per 1000 packs	6200	6900
Content of agent A gm per 100 packs	0.500	0.600
Production expenses		
\$ per 1000 packs		
Manpower, hours	90.0	100.0
Equipment, hours	40.0	50.0
Operational cost, \$	700	800

There are two kinds of raw materials, RawI and RawII, which can be used as sources of the active agent. The related data is as follows:

Raw material	Purchasing price,	Content of agent A,
RawI	100.00	0.01
RawII	199.90	0.02

Finally, the monthly resources dedicated to producing the drugs are as follows:

Budget, '\$	Manpower	Equipment	Capacity of raw materials
100000	2000	800	1000

The problem is to find the production plan which maximizes the profit of the company, i.e. minimize the purchasing and operational costs

$$100 \cdot \text{RawI} + 199.90 \cdot \text{RawII} + 700 \cdot \text{DrugI} + 800 \cdot \text{DrugII}$$

and maximize the income

$$6200 \cdot \text{DrugI} + 6900 \cdot \text{DrugII}$$

The problem can be stated as the following linear programming program:

Minimize

$$- \{100 \cdot \text{RawI} + 199.90 \cdot \text{RawII} + 700 \cdot \text{DrugI} + 800 \cdot \text{DrugII}\} + \{6200 \cdot \text{DrugI} + 6900 \cdot \text{DrugII}\} \quad (11.21)$$

subject to

$$\begin{aligned}
0.01 \cdot \text{RawI} + 0.02 \cdot \text{RawII} - 0.500 \cdot \text{DrugI} - 0.600 \cdot \text{DrugII} &\geq 0 & (a) \\
\text{RawI} + \text{RawII} &\leq 1000 & (b) \\
90.0 \cdot \text{DrugI} + 100.0 \cdot \text{DrugII} &\leq 2000 & (c) \\
40.0 \cdot \text{DrugI} + 50.0 \cdot \text{DrugII} &\leq 800 & (d) \\
100.0 \cdot \text{RawI} + 199.90 \cdot \text{RawII} + 700 \cdot \text{DrugI} + 800 \cdot \text{DrugII} &\leq 100000 & (d) \\
\text{RawI}, \text{RawII}, \text{DrugI}, \text{DrugII} &\geq 0 & (e)
\end{aligned}$$

where the variables are the amounts RawI, RawII (in kg) of raw materials to be purchased and the amounts DrugI, DrugII (in 1000 of packs) of drugs to be produced. The objective (11.21) denotes the profit to be maximized, and the inequalities can be interpreted as follows:

- Balance of the active agent.
- Storage restriction.
- Manpower restriction.
- Equipment restriction.
- Budget restriction.

Listing 11.11 is the MATLAB script which specifies the problem and solves it using the **MOSEK** optimization toolbox:

Listing 11.11: Script *rlo1.m*.

```
function rlo1()

prob.c = [-100;-199.9;6200-700;6900-800];
prob.a = sparse([0.01,0.02,-0.500,-0.600;1,1,0,0;
                0,0,90.0,100.0;0,0,40.0,50.0;100.0,199.9,700,800]);
prob.blc = [0;-inf;-inf;-inf;-inf];
prob.buc = [inf;1000;2000;800;100000];
prob.blx = [0;0;0;0];
prob.bux = [inf;inf;inf;inf];
[r,res] = mosekopt('maximize',prob);
xx      = res.sol.itr.xx;
RawI    = xx(1);
RawII   = xx(2);
DrugI   = xx(3);
DrugII  = xx(4);

disp(sprintf('*** Optimal value: %8.3f',prob.c'*xx));
disp('*** Optimal solution:');
disp(sprintf('RawI:    %8.3f',RawI));
disp(sprintf('RawII:   %8.3f',RawII));
disp(sprintf('DrugI:    %8.3f',DrugI));
disp(sprintf('DrugII:   %8.3f',DrugII));
```

When executing this script, the following is displayed:

Listing 11.12: Output of script *rlo1.m*

```
*** Optimal value: 8819.658
*** Optimal solution:
RawI:    0.000
RawII:  438.789
DrugI:   17.552
DrugII:  0.000
```

We see that the optimal solution promises the company a modest but quite respectful profit of 8.8%. Please note that at the optimal solution the balance constraint is active: the production process utilizes the full amount of the active agent contained in the raw materials.

11.3.2 Data Uncertainty and its Consequences.

Please note that not all problem data can be regarded as *absolutely* reliable; e.g. one can hardly believe that the contents of the active agent in the raw materials are *exactly* the *nominal data* 0.01 gm/kg for **RawI** and 0.02 gm/kg for **RawII**. In reality, these contents definitely vary around the indicated values. A natural assumption here is that the actual contents of the active agent a_i in **RawI** and a_{II} in **RawII** are realizations of random variables somehow distributed around the *nominal contents* $a_i^n = 0.01$ and $a_{II}^n = 0.02$. To be more specific, assume that a_i drifts in the 0.5% margin of a_i^n , i.e. it takes with probability 0.5 the values from the interval $a_i^n(1 \pm 0.005) = a_i^n\{0.00995; 0.01005\}$. Similarly, assume that a_{II} drifts in the 2% margin of a_{II}^n , taking with probabilities 0.5 the values $a_{II}^n(1 \pm 0.02) = a_{II}^n\{0.0196; 0.0204\}$. How do the perturbations of the contents of the active agent affect the production process?

The optimal solution prescribes to purchase 438.8 kg of **RawII** and to produce 17552 packs of DrugI. With the above random fluctuations in the content of the active agent in **RawII**, this production plan, with probability 0.5, will be infeasible – with this probability, the actual content of the active agent in the raw materials will be less than required to produce the planned amount of DrugI. For the sake of simplicity, assume that this difficulty is resolved in the simplest way: when the actual content of the active agent in the raw materials is insufficient, the output of the drug is reduced accordingly. With this policy, the actual production of DrugI becomes a random variable which takes, with probabilities 0.5, the nominal value of 17552 packs and the 2% less value of 17201 packs. These 2% fluctuations in the production affect the profit as well; the latter becomes a random variable taking, with probabilities 0.5, the nominal value 8,820 and the 21% less value 6,929. The expected profit is 7,843, which is by 11% less than the nominal profit 8,820 promised by the optimal solution of the problem.

We see that in our toy example that small (and in reality unavoidable) perturbations of the data may make the optimal solution infeasible, and a straightforward adjustment to the actual solution values may heavily affect the solution quality.

It turns out that the outlined phenomenon is found in many linear programs of practical origin. Usually, in these programs at least part of the data is not known exactly and can vary around its nominal values, and these data perturbations can make the nominal optimal solution – the one corresponding to the nominal data – infeasible. It turns out that the consequences of data uncertainty can be much more severe than in our toy example. The analysis of linear optimization problems from the NETLIB collection¹ reported in [BTN00] demonstrates that for 13 of 94 NETLIB problems, already 0.01% perturbations of “clearly uncertain” data can make the nominal optimal solution severely infeasible: with these perturbations, the solution, with a non-negligible probability, violates some of the constraints by 50% and more. It should be added that in the general case, in contrast to the toy example we have considered, there is no evident way to adjust the optimal solution by a small modification to the actual values of the data. Moreover there are cases when such an adjustment is impossible — in order to become feasible for the perturbed data, the nominal optimal solution should be *completely reshaped*.

11.3.3 Robust Linear Optimization Methodology

A natural approach to handling data uncertainty in optimization is offered by the *Robust Optimization Methodology* which, as applied to linear optimization, is as follows.

Uncertain Linear Programs and their Robust Counterparts.

Consider a linear optimization problem

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && l_c \leq Ax \leq u_c, \\ & && l_x \leq x \leq u_x, \end{aligned} \tag{11.22}$$

with the data $(c, A, l_c, u_c, l_x, u_x)$, and assume that this data is not known exactly; all we know is that the data varies in a given *uncertainty set* \mathcal{U} . The simplest example is the one of *interval uncertainty*, where every data entry can run through a given interval:

$$\begin{aligned} \mathcal{U} = \{ & (c, A, l_c, u_c, l_x, u_x) : \\ & (c^n - dc, A^n - dA, l_c^n - dl_c, u_c^n - du_c, l_x^n - dl_x, u_x^n - du_x) \leq (c, A, l_c, u_c, l_x, u_x) \\ & \leq (c^n + dc, A^n + dA, l_c^n + dl_c, u_c^n + du_c, l_x^n + dl_x, u_x^n + du_x) \}. \end{aligned} \tag{11.23}$$

¹ NETLIB is a collection of LP's, mainly of the real world origin, which is a standard benchmark for evaluating LP algorithms

Here

$$(c^n, A^n, l_c^n, u_c^n, l_x^n, u_x^n)$$

is the *nominal data*,

$$dc, dA, dl_c, du_c, dl_x, du_x \geq 0$$

is the *data perturbation bounds*. Please note that some of the entries in the data perturbation bounds can be zero, meaning that the corresponding data entries are certain (the expected values equals the actual values).

- The family of instances (11.22) with data running through a given uncertainty set \mathcal{U} is called an *uncertain linear optimization problem*.
- Vector x is called a *robust feasible solution* to an uncertain linear optimization problem, if it remains feasible for all realizations of the data from the uncertainty set, i.e. if

$$l_c \leq Ax \leq u_c, l_x \leq x \leq u_x$$

for all

$$(c, A, l_c, u_c, l_x, u_x) \in \mathcal{U}.$$

- If for some value t we have $c^T x \leq t$ for all realizations of the objective from the uncertainty set, we say that *robust value of the objective* at x does not exceed t .

The Robust Optimization methodology proposes to associate with an uncertain linear program its *robust counterpart* (RC) which is *the problem of minimizing the robust optimal value over the set of all robust feasible solutions*, i.e. the problem

$$\min_{t,x} \{t : c^T x \leq t, l_c \leq Ax \leq u_c, l_x \leq x \leq u_x \forall (c, A, l_c, u_c, l_x, u_x) \in \mathcal{U}\}. \quad (11.24)$$

The optimal solution to (11.24) is treated as the *uncertainty-immuned* solution to the original uncertain linear programming program.

Robust Counterpart of an Uncertain Linear Optimization Problem with Interval Uncertainty

In general, the RC (11.24) of an uncertain linear optimization problem is not a linear optimization problem since (11.24) has infinitely many linear constraints. There are, however, cases when (11.24) can be rewritten equivalently as a linear programming program; in particular, this is the case for interval uncertainty (11.23). Specifically, in the case of (11.23), the robust counterpart of uncertain linear program is equivalent to the following linear program in variables x, y, t :

$$\begin{array}{llll} \text{minimize} & t & & \\ \text{subject to} & (c^n)^T x + (dc)^T y - t \leq 0, & (a) \\ & (A^n)x - (dA)y, & (b) \\ & (A^n)x + (dA)y \leq u_c^n - du_c, & (c) \\ & 0 \leq x + y, & (d) \\ & 0 \leq -x + y, & (e) \\ & l_x^n + dl_x \leq x \leq u_x^n - du_x, & (f) \end{array} \quad (11.25)$$

The origin of (11.25) is quite transparent: The constraints $d - e$ in (11.25) linking x and y merely say that $y_i \geq |x_i|$ for all i . With this in mind, it is evident that at every feasible solution to (11.25) the entries in the vector

$$(A^n)x - (dA)y$$

are lower bounds on the entries of Ax with A from the uncertainty set (11.23), so that (b) in (11.25) ensures that $l_c \leq Ax$ for all data from the uncertainty set. Similarly, (c), (a) and f in (11.25) ensure, for

all data from the uncertainty set, that $Ax \leq u_c$, $c^T x \leq t$, and that the entries in x satisfy the required lower and upper bounds, respectively.

Please note that at the optimal solution to (11.25), one clearly has $y_j = |x_j|$. It follows that when the bounds on the entries of x impose nonnegativity (nonpositivity) of an entry x_j , then there is no need to introduce the corresponding additional variable y_i — from the very beginning it can be replaced with x_j , if x_j is nonnegative, or with $-x_j$, if x_j is nonpositive.

Another possible formulation of problem (11.25) is the following. Let

$$l_c^n + dl_c = (A^n)x - (dA)y - f, f \geq 0$$

then this equation is equivalent to (a) – (b) in (11.25). If $(l_c)_i = -\infty$, then equation i should be dropped from the computations. Similarly,

$$-x + y = g \geq 0$$

is equivalent to (d) in (11.25). This implies that

$$l_c^n + dl_c - (A^n)x + f = -(dA)y$$

and that

$$y = g + x$$

Substituting these values into (11.25) gives

$$\begin{array}{llll} \text{minimize} & & t & \\ \text{subject to} & (c^n)^T x + (dc)^T (g + x) - t & \leq & 0, \\ & 0 & \leq & f, \\ & 2(A^n)x + (dA)(g + x) + f + l_c^n + dl_c & \leq & u_c^n - du_c, \\ & 0 & \leq & g, \\ & 0 & \leq & 2x + g, \\ & l_x^n + dl_x & \leq & x \leq u_x^n - du_x, \end{array}$$

which after some simplifications leads to

$$\begin{array}{llllll} \text{minimize} & & t & & & \\ \text{subject to} & (c^n + dc)^T x + (dc)^T g - t & \leq & 0, & (a) \\ & 0 & \leq & f, & (b) \\ & 2(A^n + dA)x + (dA)g + f - (l_c^n + dl_c) & \leq & u_c^n - du_c, & (c) \\ & 0 & \leq & g, & (d) \\ & 0 & \leq & 2x + g, & (e) \\ & l_x^n + dl_x & \leq & x & \leq u_x^n - du_x, & (f) \end{array}$$

and

$$\begin{array}{llllll} \text{minimize} & & t & & & \\ \text{subject to} & (c^n + dc)^T x + (dc)^T g - t & \leq & 0, & (a) \\ & 2(A^n + dA)x + (dA)g + f & \leq & u_c^n - du_c + l_c^n + dl_c, & (b) \\ & 0 & \leq & 2x + g, & (c) \\ & 0 & \leq & f, & (d) \\ & 0 & \leq & g, & (e) \\ & l_x^n + dl_x & \leq & x & \leq u_x^n - du_x. & (f) \end{array} \quad (11.26)$$

Please note that this problem has more variables but much fewer constraints than (11.25). Therefore, (11.26) is likely to be solved faster than (11.25). Note too that (11.26).b is trivially redundant if $l_x^n + dl_x \geq 0$.

Introductory Example (continued)

Let us apply the Robust Optimization methodology to our drug production example presented in Sec. 11.3.1, assuming that the only uncertain data is the contents of the active agent in the raw materials,

and that these contents vary in 0.5% and 2% neighborhoods of the respective nominal values 0.01 and 0.02. With this assumption, the problem becomes an uncertain LP affected by interval uncertainty; the robust counterpart (11.25) of this uncertain LP is the linear program

$$\begin{aligned}
& \text{(Drug_RC) :} \\
& \text{maximize} \\
& t \\
& \text{subject to} \\
& t \leq -100 \cdot \text{RawI} - 199.9 \cdot \text{RawII} + 5500 \cdot \text{DrugI} + 6100 \cdot \text{DrugII} \\
& 0.01 \cdot 0.995 \cdot \text{RawI} + 0.02 \cdot 0.98 \cdot \text{RawII} - 0.500 \cdot \text{DrugI} - 0.600 \cdot \text{DrugII} \geq 0 \\
& \text{RawI} + \text{RawII} \leq 1000 \\
& 90.0 \cdot \text{DrugI} + 100.0 \cdot \text{DrugII} \leq 2000 \\
& 40.0 \cdot \text{DrugI} + 50.0 \cdot \text{DrugII} \leq 800 \\
& 100.0 \cdot \text{RawI} + 199.90 \cdot \text{RawII} + 700 \cdot \text{DrugI} + 800 \cdot \text{DrugII} \leq 100000 \\
& \text{RawI}, \text{RawII}, \text{DrugI}, \text{DrugII} \geq 0
\end{aligned} \tag{11.27}$$

Solving this problem with **MOSEK** we get the following output:

Listing 11.13: Output solving problem (11.27).

```

*** Optimal value: 8294.567
*** Optimal solution:
RawI:      877.732
RawII:      0.000
DrugI:      17.467
DrugII:      0.000

```

We see that the robust optimal solution we have built *costs money* – it promises a profit of just 8,295 (cf. with the profit of 8,820 promised by the nominal optimal solution). Please note, however, that the robust optimal solution remains feasible whatever are the realizations of the uncertain data from the uncertainty set in question, while the nominal optimal solution requires adjustment to this data and, with this adjustment, results in the average profit of 7,843, which is by 5.4% *less* than the profit of ‘8,295 *guaranteed* by the robust optimal solution. Note too that the robust optimal solution is significantly different from the nominal one: both solutions prescribe to produce the same drug **DrugI** (in the amounts 17,467 and 17,552 packs, respectively) but from different raw materials, **RawI** in the case of the robust solution and **RawII** in the case of the nominal solution. The reason is that although the price per unit of the active agent for **RawII** is slightly less than for **RawI**, the content of the agent in **RawI** is more stable, so when possible fluctuations of the contents are taken into account, **RawI** turns out to be more profitable than **RawII**.

11.3.4 Random Uncertainty and Ellipsoidal Robust Counterpart

In some cases, it is natural to assume that the perturbations affecting different uncertain data entries are random and independent of each other. In these cases, the robust counterpart based on the interval model of uncertainty seems to be too conservative: Why should we expect that all the data will be simultaneously driven to its most unfavorable values and immune the solution against this highly unlikely situation? A less conservative approach is offered by the *ellipsoidal* model of uncertainty. To motivate this model, let us see what happens with a particular linear constraint

$$a^T x \leq b \tag{11.28}$$

at a given candidate solution x in the case when the vector a of coefficients of the constraint is affected by random perturbations:

$$a = a^n + \zeta, \tag{11.29}$$

where a^n is the vector of nominal coefficients and ζ is a random perturbation vector with zero mean and covariance matrix V_a . In this case the value of the left-hand side of (11.28), evaluated at a given x , becomes a random variable with the expected value $(a^n)^T x$ and the standard deviation $\sqrt{x^T V_a x}$. Now let us act as an engineer who believes that the value of a random variable never exceeds its mean plus 3 times the standard deviation; we do not intend to be that specific and replace 3 in the above rule by

a safety parameter Ω which will be in our control. Believing that the value of a random variable *never* exceeds its mean plus Ω times the standard deviation, we conclude that a *safe* version of (11.28) is the inequality

$$(a^n)^T x + \Omega \sqrt{x^T V_a x} \leq b. \quad (11.30)$$

The word *safe* above admits a quantitative interpretation: If x satisfies (11.30), one can bound from above the probability of the event that random perturbations (11.29) result in violating the constraint (11.28) evaluated at x . The bound in question depends on what we know about the distribution of ζ , e.g.

- We always have the bound given by the Tschebyshev inequality: x satisfies (11.30) \Rightarrow

$$\text{Prob}\{a^T x > b\} \leq \frac{1}{\Omega^2}.$$

- When ζ is Gaussian, then the Tschebyshev bound can be improved to: x satisfies (11.30) \Rightarrow

$$\text{Prob}\{a^T x > b\} \leq \frac{1}{\sqrt{2\pi}} \int_{\Omega}^{\infty} \exp\{-t^2/2\} dt \leq 0.5 \exp\{-\Omega^2/2\}. \quad (11.31)$$

- Assume that $\zeta = D\xi$, where Δ is certain $n \times m$ matrix, and $\xi = (\xi_1, \dots, \xi_m)^T$ is a random vector with independent coordinates ξ_1, \dots, ξ_m symmetrically distributed in the segment $[-1, 1]$. Setting $V = DD^T$ (V is a natural *upper bound* on the covariance matrix of ζ), one has: x satisfies (11.30) implies

$$\text{Prob}\{a^T x > b\} \leq 0.5 \exp\{-\Omega^2/2\}. \quad (11.32)$$

Please note that in order to ensure the bounds in (11.31) and (11.32) to be $\leq 10^{-6}$, it suffices to set $\Omega = 5.13$.

Now, assume that we are given a linear program affected by random perturbations:

$$\begin{aligned} & \text{minimize} && [c^n + dc]^T x \\ & \text{subject to} && (l_c)_i \leq [a_i^n + da_i]^T x \leq (u_c)_i, i = 1, \dots, m, \\ & && l_x \leq x \leq u_x, \end{aligned} \quad (11.33)$$

where $(c^n, \{a_i^n\}_{i=1}^m, l_c, u_c, l_x, u_x)$ are the nominal data, and dc, da_i are random perturbations with zero means³. Assume, for the sake of definiteness, that every one of the random perturbations dc, da_1, \dots, da_m satisfies either the assumption of item 2 or the assumption of item 3, and let V_c, V_1, \dots, V_m be the corresponding (upper bounds on the) covariance matrices of the perturbations. Choosing a safety parameter Ω and replacing the objective and the bodies of all the constraints by their safe bounds as explained above, we arrive at the following optimization problem:

$$\begin{aligned} & \text{minimize} && t \\ & \text{subject to} && [c^n]^T x + \Omega \sqrt{x^T V_c x} \leq t, \\ & && (l_c)_i \leq [a_i^n]^T x - \Omega \sqrt{x^T V_{a_i} x} \leq [a_i^n]^T x + \Omega \sqrt{x^T V_{a_i} x} \leq (u_c)_i, i = 1, \dots, m, \\ & && l_x \leq x \leq u_x. \end{aligned} \quad (11.34)$$

The relation between problems (11.34) and (11.33) is as follows:

- If (x, t) is a feasible solution of (11.34), then with probability at least

$$p = 1 - (m+1) \exp\{-\Omega^2/2\}$$

x is feasible for randomly perturbed problem (11.33), and t is an upper bound on the objective of (11.33) evaluated at x .

- We see that if Ω is not too small (11.34) can be treated as a “safe version” of (11.33).

³ For the sake of simplicity, we assume that the bounds l_c, u_c, l_x, u_x are not affected by uncertainty; extensions to the case when it is not so are evident.

On the other hand, it is easily seen that (11.34) is nothing but the robust counterpart of the uncertain linear optimization problem with the nominal data $(c^n, \{a_i^n\}_{i=1}^m, l_c, u_c, l_x, u_x)$ and the row-wise ellipsoidal uncertainty given by the matrices $V_c, V_{a_1}, \dots, V_{a_m}$. In the corresponding uncertainty set, the uncertainty affects the coefficients of the objective and the constraint matrix only, and the perturbation vectors affecting the objective and the vectors of coefficients of the linear constraints run, independently of each other, through the respective ellipsoids

$$\begin{aligned} E_c &= \left\{ dc = \Omega V_c^{1/2} u : u^T u \leq 1 \right\} \\ E_{a_i} &= \left\{ da_i = \Omega V_{a_i}^{1/2} u : u^T u \leq 1 \right\}, i = 1, \dots, m. \end{aligned}$$

It turns out that in many cases the ellipsoidal model of uncertainty is significantly less conservative and thus better suited for practice, than the interval model of uncertainty.

Last but not least, it should be mentioned that problem (11.34) is equivalent to a conic quadratic program, specifically to the program

$$\begin{aligned} &\text{minimize} && t \\ &\text{subject to} && [c^n]^T x + \Omega z \leq t, \\ & && (l_c)_i \leq [a_i^n]^T x - \Omega z_i, \\ & && [a_i^n]^T x + \Omega z_i \leq (u_c)_i, i = 1, \dots, m, \\ & && 0 = w - D_c x \\ & && 0 = w^i - D_{a_i} x, \quad i = 1, \dots, m, \\ & && 0 \leq z - \sqrt{w^T w}, \\ & && 0 \leq z_i - \sqrt{(w^i)^T w^i}, \quad i = 1, \dots, m, \\ & && l_x \leq x \leq u_x. \end{aligned}$$

where D_c and D_{a_i} are matrices satisfying the relations

$$V_c = D_c^T D_c, V_{a_i} = D_{a_i}^T D_{a_i}, i = 1, \dots, m.$$

Example: Interval and Ellipsoidal Robust Counterparts of Uncertain Linear Constraint with Independent Random Perturbations of Coefficients

Consider a linear constraint

$$l \leq \sum_{j=1}^n a_j x_j \leq u \quad (11.35)$$

and assume that the a_j coefficients of the body of the constraint are uncertain and vary in intervals $a_j^n \pm \sigma_j$. The worst-case-oriented model of uncertainty here is the interval one, and the corresponding robust counterpart of the constraint is given by the system of linear inequalities

$$\begin{aligned} l &\leq \sum_{j=1}^n a_j^n x_j - \sum_{j=1}^n \sigma_j y_j, \\ &\quad \sum_{j=1}^n a_j^n x_j + \sum_{j=1}^n \sigma_j y_j \leq u, \\ 0 &\leq x_j + y_j, \\ 0 &\leq -x_j + y_j, \quad j = 1, \dots, n. \end{aligned} \quad (11.36)$$

Now, assume that we have reasons to believe that the true values of the coefficients a_j are obtained from their nominal values a_j^n by random perturbations, independent for different j and symmetrically distributed in the segments $[-\sigma_j, \sigma_j]$. With this assumption, we are in the situation of item 3 and can replace the uncertain constraint (11.35) with its ellipsoidal robust counterpart

$$\begin{aligned} l &\leq \sum_{j=1}^n a_j^n x_j - \Omega z, \\ &\quad \sum_{j=1}^n a_j^n x_j + \Omega z \leq u, \\ 0 &\leq z - \sqrt{\sum_{j=1}^n \sigma_j^2 x_j^2}. \end{aligned} \quad (11.37)$$

Please note that with the model of random perturbations, a vector x satisfying (11.37) satisfies a realization of (11.35) with probability at least $1 - \exp\{-\Omega^2/2\}$; for $\Omega = 6$. This probability is $\geq 1 - 1.5 \cdot 10^{-8}$,

which for all practical purposes is the same as saying that x satisfies all realizations of (11.35). On the other hand, the uncertainty set associated with (11.36) is the box

$$B = \{a = (a_1, \dots, a_n)^T : a_j^n - \sigma_j \leq a_j \leq a_j^n + \sigma_j, j = 1, \dots, n\},$$

while the uncertainty set associated with (11.37) is the ellipsoid

$$E(\Omega) = \left\{ a = (a_1, \dots, a_n)^T : \sum_{j=1}^n (a_j - a_j^n)^2 \frac{2}{\sigma_j^2} \leq \Omega^2 \right\}.$$

For a moderate value of Ω , say $\Omega = 6$, and $n \geq 40$, the ellipsoid $E(\Omega)$ in its diameter, typical linear sizes, volume, etc. is incomparably less than the box B , the difference becoming more dramatic the larger the dimension n of the box and the ellipsoid. It follows that the ellipsoidal robust counterpart (11.37) of the randomly perturbed uncertain constraint (11.35) is much less conservative than the interval robust counterpart (11.36), while ensuring basically the same “robustness guarantees”. To illustrate this important point, consider the following numerical examples:

There are n different assets on the market. The return on 1 invested in asset j is a random variable distributed symmetrically in the segment $[\delta_j - \sigma_j, \delta_j + \sigma_j]$, and the returns on different assets are independent of each other. The problem is to distribute ‘1’ among the assets in order to get the largest possible total return on the resulting portfolio.

A natural model of the problem is an uncertain linear optimization problem

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^n a_j x_j \\ \text{subject to} & \sum_{j=1}^n x_j = 1, \\ & 0 \leq x_j, \quad j = 1, \dots, n. \end{array}$$

where a_j are the uncertain returns of the assets. Both the nominal optimal solution (set all returns a_j equal to their nominal values δ_j) and the risk-neutral Stochastic Programming approach (maximize the expected total return) result in the same solution: Our money should be invested in the most promising asset(s) – the one(s) with the maximal nominal return. This solution, however, can be very unreliable if, as is typically the case in reality, the most promising asset has the largest volatility σ and is in this sense the most risky. To reduce the risk, one can use the Robust Counterpart approach which results in the following optimization problems.

The Interval Model of Uncertainty:

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & 0 \leq -t + \sum_{j=1}^n (\delta_j - \sigma_j) x_j, \\ & \sum_{j=1}^n x_j = 1, \\ & 0 \leq x_j, \quad j = 1, \dots, n \end{array} \quad (11.38)$$

and

The ellipsoidal Model of Uncertainty:}

$$\begin{array}{ll} \text{maximize} & t \\ \text{subject to} & 0 \leq -t + \sum_{j=1}^n (\delta_j) x_j - \Omega z, \\ & 0 \leq z - \sqrt{\sum_{j=1}^n \sigma_j^2 x_j^2}, \\ & \sum_{j=1}^n x_j = 1, \\ & 0 \leq x_j, \quad j = 1, \dots, n. \end{array} \quad (11.39)$$

Note that the problem (11.39) is essentially the risk-averted portfolio model proposed in mid-50’s by Markowitz.

The solution of (11.38) is evident — our ‘1’ should be invested in the asset(s) with the largest possible *guaranteed* return $\delta_j - \sigma_j$. In contrast to this very conservative policy (which in reality prescribes to keep

"MSK_CT_PEXP"
A primal exponential cone.

"MSK_CT_DEXP"
A dual exponential cone.

"MSK_CT_PPOW"
A primal power cone.

"MSK_CT_DPOW"
A dual power cone.

"MSK_CT_ZERO"
The zero cone.

nametype
Name types

"MSK_NAME_TYPE_GEN"
General names. However, no duplicate and blank names are allowed.

"MSK_NAME_TYPE_MPS"
MPS type names.

"MSK_NAME_TYPE_LP"
LP type names.

scopr
SCopt operator types

"MSK_OPR_ENT"
Entropy

"MSK_OPR_EXP"
Exponential

"MSK_OPR_LOG"
Logarithm

"MSK_OPR_POW"
Power

"MSK_OPR_SQRT"
Square root

symmattype
Cone types

"MSK_SYMMAT_TYPE_SPARSE"
Sparse symmetric matrix.

dataformat
Data format types

"MSK_DATA_FORMAT_EXTENSION"
The file extension is used to determine the data file format.

"MSK_DATA_FORMAT_MPS"
The data file is MPS formatted.

"MSK_DATA_FORMAT_LP"
The data file is LP formatted.

"MSK_DATA_FORMAT_OP"
The data file is an optimization problem formatted file.

"MSK_DATA_FORMAT_FREE_MPS"
The data a free MPS formatted file.

"MSK_DATA_FORMAT_TASK"
Generic task dump file.

"MSK_DATA_FORMAT_PTF"
(P)retty (T)ext (F)format.

"MSK_DATA_FORMAT_CB"
Conic benchmark format,


```

"MSK_PAR_INT_TYPE"
    Is an integer parameter.
"MSK_PAR_STR_TYPE"
    Is a string parameter.
problemitem
    Problem data items
"MSK_PI_VAR"
    Item is a variable.
"MSK_PI_CON"
    Item is a constraint.
"MSK_PI_CONE"
    Item is a cone.
problemtypes
    Problem types
"MSK_PROBTYPE_LO"
    The problem is a linear optimization problem.
"MSK_PROBTYPE_QO"
    The problem is a quadratic optimization problem.
"MSK_PROBTYPE_QCQO"
    The problem is a quadratically constrained optimization problem.
"MSK_PROBTYPE_CONIC"
    A conic optimization.
"MSK_PROBTYPE_MIXED"
    General nonlinear constraints and conic constraints. This combination can not be solved by
    MOSEK.
prosta
    Problem status keys
"MSK_PRO_STA_UNKNOWN"
    Unknown problem status.
"MSK_PRO_STA_PRIM_AND_DUAL_FEAS"
    The problem is primal and dual feasible.
"MSK_PRO_STA_PRIM_FEAS"
    The problem is primal feasible.
"MSK_PRO_STA_DUAL_FEAS"
    The problem is dual feasible.
"MSK_PRO_STA_PRIM_INFEAS"
    The problem is primal infeasible.
"MSK_PRO_STA_DUAL_INFEAS"
    The problem is dual infeasible.
"MSK_PRO_STA_PRIM_AND_DUAL_INFEAS"
    The problem is primal and dual infeasible.
"MSK_PRO_STA_ILL_POSED"
    The problem is ill-posed. For example, it may be primal and dual feasible but have a positive
    duality gap.
"MSK_PRO_STA_PRIM_INFEAS_OR_UNBOUNDED"
    The problem is either primal infeasible or unbounded. This may occur for mixed-integer
    problems.
xmlwriteroutputtype
    XML writer output mode
"MSK_WRITE_XML_MODE_ROW"
    Write in row order.

```


"MSK_SOL_ITEM_XC"
Solution for the constraints.

"MSK_SOL_ITEM_XX"
Variable solution.

"MSK_SOL_ITEM_Y"
Lagrange multipliers for equations.

"MSK_SOL_ITEM_SLC"
Lagrange multipliers for lower bounds on the constraints.

"MSK_SOL_ITEM_SUC"
Lagrange multipliers for upper bounds on the constraints.

"MSK_SOL_ITEM_SLX"
Lagrange multipliers for lower bounds on the variables.

"MSK_SOL_ITEM_SUX"
Lagrange multipliers for upper bounds on the variables.

"MSK_SOL_ITEM_SNX"
Lagrange multipliers corresponding to the conic constraints on the variables.

solsta
Solution status keys

"MSK_SOL_STA_UNKNOWN"
Status of the solution is unknown.

"MSK_SOL_STA_OPTIMAL"
The solution is optimal.

"MSK_SOL_STA_PRIM_FEAS"
The solution is primal feasible.

"MSK_SOL_STA_DUAL_FEAS"
The solution is dual feasible.

"MSK_SOL_STA_PRIM_AND_DUAL_FEAS"
The solution is both primal and dual feasible.

"MSK_SOL_STA_PRIM_INFEAS_CER"
The solution is a certificate of primal infeasibility.

"MSK_SOL_STA_DUAL_INFEAS_CER"
The solution is a certificate of dual infeasibility.

"MSK_SOL_STA_PRIM_ILLPOSED_CER"
The solution is a certificate that the primal problem is illposed.

"MSK_SOL_STA_DUAL_ILLPOSED_CER"
The solution is a certificate that the dual problem is illposed.

"MSK_SOL_STA_INTEGER_OPTIMAL"
The primal solution is integer optimal.

solttype
Solution types

"MSK_SOL_BAS"
The basic solution.

"MSK_SOL_ITR"
The interior solution.

"MSK_SOL_ITG"
The integer solution.

solveform
Solve primal or dual form

"MSK_SOLVE_FREE"
The optimizer is free to solve either the primal or the dual problem.

"MSK_SOLVE_PRIMAL"
The optimizer should solve the primal problem.

"MSK_SOLVE_DUAL"
The optimizer should solve the dual problem.

stakey
Status keys

"MSK_SK_UNK"
The status for the constraint or variable is unknown.

"MSK_SK_BAS"
The constraint or variable is in the basis.

"MSK_SK_SUPBAS"
The constraint or variable is super basic.

"MSK_SK_LOW"
The constraint or variable is at its lower bound.

"MSK_SK_UPR"
The constraint or variable is at its upper bound.

"MSK_SK_FIX"
The constraint or variable is fixed.

"MSK_SK_INF"
The constraint or variable is infeasible in the bounds.

startpointtype
Starting point types

"MSK_STARTING_POINT_FREE"
The starting point is chosen automatically.

"MSK_STARTING_POINT_GUESS"
The optimizer guesses a starting point.

"MSK_STARTING_POINT_CONSTANT"
The optimizer constructs a starting point by assigning a constant value to all primal and dual variables. This starting point is normally robust.

"MSK_STARTING_POINT_SATISFY_BOUNDS"
The starting point is chosen to satisfy all the simple bounds on nonlinear variables. If this starting point is employed, then more care than usual should be employed when choosing the bounds on the nonlinear variables. In particular very tight bounds should be avoided.

streamtype
Stream types

"MSK_STREAM_LOG"
Log stream. Contains the aggregated contents of all other streams. This means that a message written to any other stream will also be written to this stream.

"MSK_STREAM_MSG"
Message stream. Log information relating to performance and progress of the optimization is written to this stream.

"MSK_STREAM_ERR"
Error stream. Error messages are written to this stream.

"MSK_STREAM_WRN"
Warning stream. Warning messages are written to this stream.

value
Integer values

"MSK_MAX_STR_LEN"
Maximum string length allowed in **MOSEK**.

"MSK_LICENSE_BUFFER_LENGTH"
The length of a license key buffer.

variabletype
Variable types

This log-sum-exp bound is equivalent to

$$\sum_i \exp(a_i^T x + b_i - t) \leq 1$$

and requires bounding each exponential function as explained above.

Dual geometric optimization (DGopt)

The objective function of a dual geometric problem involves maximizing expressions of the form

$$x \log \frac{c}{x} \quad \text{and} \quad x_i \log \frac{e^T x}{x_i},$$

which can be achieved using bounds $t \leq x \log \frac{y}{x}$, that is $(t, x, y) \in K_{\text{exp}}$.

[integer]

This contains a space-separated list of variables and defines the constraint that the listed variables must be integer-valued.

[hints]

This may contain only non-essential data; for example estimates of the number of variables, constraints and non-zeros. Placed before all other sections containing data this may reduce the time spent reading the file.

In the `hints` section, any subsection which is not recognized by **MOSEK** is simply ignored. In this section a hint is defined as follows:

```
[hint ITEM] value [/hint]
```

The hints recognized by **MOSEK** are:

- `numvar` (number of variables),
- `numcon` (number of linear/quadratic constraints),
- `numanz` (number of linear non-zeros in constraints),
- `numqnz` (number of quadratic non-zeros in constraints).

[solutions]

This section can contain a set of full or partial solutions to a problem. Each solution must be specified using a `[solution]`-section, i.e.

```
[solutions]
[solution]...[/solution] #solution 1
[solution]...[/solution] #solution 2
#other solutions....
[solution]...[/solution] #solution n
[/solutions]
```

The syntax of a `[solution]`-section is the following:

```
[solution SOLTYPE status=STATUS]...[/solution]
```

where `SOLTYPE` is one of the strings

- `interior`, a non-basic solution,
- `basic`, a basic solution,
- `integer`, an integer solution,

and `STATUS` is one of the strings

- `UNKNOWN`,
- `OPTIMAL`,
- `INTEGER_OPTIMAL`,
- `PRIM_FEAS`,
- `DUAL_FEAS`,
- `PRIM_AND_DUAL_FEAS`,
- `NEAR_OPTIMAL`,
- `NEAR_PRIM_FEAS`,


```
[b] 0 <= * [/b]
[b] 0 <= x2 <= 10 [/b]
[/bounds]
```

Quadratic Example qo1.opf

An example of a quadratic optimization problem is

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

This can be formulated in `opf` as shown below.

Listing 16.2: Example of an OPF file for a quadratic problem.

```
[comment]
  The qo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 3 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
  [hint NUMQNZ] 4 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3
[/variables]

[objective minimize 'obj']
  # The quadratic terms are often written with a factor of 1/2 as here,
  # but this is not required.

  - x2 + 0.5 ( 2.0 x1 ^ 2 - 2.0 x3 * x1 + 0.2 x2 ^ 2 + 2.0 x3 ^ 2 )
[/objective]

[constraints]
  [con 'c1'] 1.0 <= x1 + x2 + x3 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]
```

Conic Quadratic Example cqo1.opf

Consider the example:

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

Please note that the type of the cones is defined by the parameter to `[cone ...]`; the content of the `cone`-section is the names of variables that belong to the cone. The resulting OPF file is in [Listing 16.3](#).

Listing 16.3: Example of an OPF file for a conic quadratic problem.

```
[comment]
  The cqo1 example in OPF format.
[/comment]

[hints]
  [hint NUMVAR] 6 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3 x4 x5 x6
[/variables]

[objective minimize 'obj']
  x4 + x5 + x6
[/objective]

[constraints]
  [con 'c1'] x1 + x2 + 2e+00 x3 = 1e+00 [/con]
[/constraints]

[bounds]
  # We let all variables default to the positive orthant
  [b] 0 <= * [/b]

  # ...and change those that differ from the default
  [b] x4,x5,x6 free [/b]

  # Define quadratic cone:  $x_4 \geq \sqrt{x_1^2 + x_2^2}$ 
  [cone quad 'k1'] x4, x1, x2 [/cone]

  # Define rotated quadratic cone:  $2 x_5 x_6 \geq x_3^2$ 
  [cone rquad 'k2'] x5, x6, x3 [/cone]
[/bounds]
```

Mixed Integer Example milo1.opf

Consider the mixed integer problem:

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

This can be implemented in OPF with the file in [Listing 16.4](#).

Listing 16.4: Example of an OPF file for a mixed-integer linear problem.

```
[comment]
  The milo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 2 [/hint]
  [hint NUMCON] 2 [/hint]
  [hint NUMANZ] 4 [/hint]
[/hints]
```

(continues on next page)


```

# Three scalar variables in this one conic domain:
#   | Three are free.
VAR
3 1
F 3

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

# Five coordinates in  $F^{\text{obj}}_j$  coefficients:
#   |  $F^{\text{obj}}[0][0,0] = 2.0$ 
#   |  $F^{\text{obj}}[0][1,0] = 1.0$ 
#   | 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  $a^{\text{obj}}_j$  coefficients:
#   |  $a^{\text{obj}}[1] = 1.0$ 
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

```


given,

1. $g_0^{obj} = x_0 + 0.64x_1$.
2. $g_1^{obj} = 1.11x_0 + 0.76x_1$.
3. $g_2^{obj} = 1.11x_0 + 0.85x_1$.

Its formulation in the CBF format is reported in [Listing 16.5](#).

Listing 16.5: Problem (16.14) in CBF format.

```
# File written using this version of the Conic Benchmark Format:
#   | Version 1.
VER
1

# The sense of the objective is:
#   | Maximize.
OBJSENSE
MAX

# Two scalar variables in this one conic domain:
#   | Two are nonnegative.
VAR
2 1
L+ 2

# Two scalar constraints with affine expressions in these two conic domains:
#   | One is in the nonpositive domain.
#   | One is in the nonnegative domain.
CON
2 2
L- 1
L+ 1

# Two coordinates in a^{obj}_j coefficients:
#   | a^{obj}[0] = 1.0
#   | a^{obj}[1] = 0.64
OBJCOORD
2
0 1.0
1 0.64

# Four coordinates in a_ij coefficients:
#   | a[0,0] = 50.0
#   | a[1,0] = 3.0
#   | and more...
ACCOORD
4
0 0 50.0
1 0 3.0
0 1 31.0
1 1 -2.0

# Two coordinates in b_i coefficients:
#   | b[0] = -250.0
#   | b[1] = 4.0
BCCOORD
2
0 -250.0
1 4.0
```

(continues on next page)

16.7 The JSON Format

MOSEK provides the possibility to read/write problems in valid JSON format.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

The official JSON website <http://www.json.org> provides plenty of information along with the format definition.

MOSEK defines two JSON-like formats:

- *jtask*
- *jsol*

Despite being text-based human-readable formats, *jtask* and *jsol* files will include no indentation and no new-lines, in order to keep the files as compact as possible. We therefore strongly advise to use JSON viewer tools to inspect *jtask* and *jsol* files.

16.7.1 *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**.

16.7.2 *jsol* format

It stores a problem solution. The *jsol* format contains all solutions and information items.

16.7.3 A *jtask* example

In Listing 16.6 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 16.6: 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",
```

(continues on next page)

```

        "x4"
    ],
    "bk": [
        "lo",
        "ra",
        "lo",
        "lo"
    ],
    "bl": [
        0.0,
        0.0,
        0.0,
        0.0
    ],
    "bu": [
        1e+30,
        1e+1,
        1e+30,
        1e+30
    ],
    "type": [
        "cont",
        "cont",
        "cont",
        "cont"
    ]
},
"con": {
    "name": [
        "c1",
        "c2",
        "c3"
    ],
    "bk": [
        "fx",
        "lo",
        "up"
    ],
    "bl": [
        3e+1,
        1.5e+1,
        -1e+30
    ],
    "bu": [
        3e+1,
        1e+30,
        2.5e+1
    ]
},
"objective": {
    "sense": "max",
    "name": "obj",
    "c": {
        "subj": [
            0,
            1,
            2,
            3
        ],
        "val": [
            3e+0,

```


duasen, 153
names, 150
primal_repair, 152
prisen, 152
prisen_data, 152
prob, 148
res, 149
solution, 150
solver_solutions, 150

